**RBSP ECT**

**Science Operations Center**

**DBProcessing: The Manual**

**Date: October 16th, 2013**

**Brian Larsen**

**Los Alamos National Laboratory**

**Revision - 1**

**Changes**

|  |  |  |  |
| --- | --- | --- | --- |
| **Version** | **Description of Change** | **Approved by** | **Date** |
| A  1 | Preliminary Draft Version  Updated to be a manual explaining how to perfom tasks |  | 04/24/2012  10/15/2013 |

**Table of Contents**

1.0 Purpose 3

2.0 Overview 3

2.1 Flow 4

2.2 Database creation 4

2.2.1 Configuration file specification 4

2.2.1.1 Required sections 5

2.2.1.2 Optional sections 6

2.2.2 Format 6

2.3 Inspector 7

2.3.1 Example inspector #1 8

2.3.2 Example Inspector #2 9

2.4 Database structure 10

# Purpose

Provide a manual for setup, use, and maintenance of the LANL DBProcessing processing chain.

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.

# Overview

DBProcessing is a general purpose processing chain that operations on an N-files in and 1-file out basis. The database backend keeps track of the relationship between products and how to create a child product from a (or several) parent(s).

The database in the backend is sqlite3 allowing for easy backup and deployment of the chain in whatever environment. This sqlite3 backend could be replaced by a higher performance database but this would likely yield only minimum performance increases as database calls are not the bottleneck in processing, it is the codes themselves that process data.

A single DBProcessing chain may only process data for one mission, but that mission may have multiple satellites and instruments.

## Flow

The general flow of files and processing is:

1. A new data file appears into an incoming directory by some external means
2. DBProcessing compares that file to a list of “inspectors” that chain the file assigning a product to the file
3. The file is checked against the input products for all processes, if this file (and any/all others) are present to complete the process it is run
4. The output file is placed in incoming from step 1 and the process repeats.

If the inserted file in step 1 is a new version of an existing file the flow is the same with a new version of the output file being created.

A chain is an instance of DBProcessing running with a particular configuration. Processing relies on the user to do several steps, normally performed from a computers crontab.

1. New files are placed into incoming
2. ProcessQueue.py –i is run to ingest the new files
3. ProcessQueue.py –p is run to process pending files

## Database creation

The creation of a new instance of the processing chain involves creating a configuration file to explain the relationship between products and the processes that act upon them.

### Configuration file specification

The configuration files are Microsoft Windows INI format files parsed by the python ConfigParser [[1]](#footnote-1) module. This format has sections specified inside square brackets such as [mission] and then under each section key value pairs specified either in key: value or key=value format. There exists a special DEFUALT section that populates key-value pairs to every section in the config file, this is useful for options that may be the same for all products or processes. A single configuration file may only contain one mission, satellite, and instrument. However multiple config files can be used to define the same DBProcessing chain. In addition the whole config file does not have to be written at once, applying a config file to an existing DBprocessing chain will add to the chain elements present in the config file and not in the chain, the current version of DBProcessing does not allow for item update via configuration file change.

#### Required sections

The required sections are defined and explained below:

**[mission]** – this section defines the mission fur the chain. (one per config file) (must be unique in the chain)

rootdir: full path to the base directory of the data and codes for this chain

mission\_name: the name of the mission (must be unique in the chain)

incoming\_dir: relative path from rootdir to the incoming directory

**[satellite]** – this section defines the satellite for the config file (one per config file)

satellite\_name: the name of the satellite (must be unique in the chain)

**[instrument]** – the name of the instrument for the config file (one per config file)

instrument\_name: the name of the instrument

**[product\_XXX]** – this repeated section defines each product used in the chain. Each name must be unique and start with “product\_”.

product\_name: name of the product (must be unique in the chain)

relative\_path: relative path from mission-rootdir to store this product

level: level of the data, only used to sort the order processes are run. Inputs with lower levels are run before higher ones. By convention the levels are 0, 1, 2, 3 etc.

format: format specification of the product, this is used by the chain to create an output filename for a process. See section 2.2.2 for format specification.

product\_description: human readable description of the product

inspector\_filename: filename of the inspector used to identify the product. See section XXX for inspector particulars.

inspector\_relative\_path: relative path to the inspectors based on mission-rootdir

inspector\_description: human readable description

inspector\_version: version of the inspector. See section XXX for explanation of versions.

inspector\_output\_interface: the interface version of the output produced by the inspector. See section XXX for explanation of versions.

inspector\_active: Boolean (True/False) defining if this inspector is in use.

inspector\_date\_written: date the inspector was written for recordkeeping YYYY-MM-DD

inspector\_newest\_version: Boolean (True/False) defining if this version of the inspector is the newest

inspector\_arguments: key value pairs passed through to the inspector. These are in the format key=value. Commonly product\_name={PRODUCT}

**[process\_XXX]** – this repeated section defines each process used to move products along in the chain. Each name must be unique and start with “process\_”.

process\_name: name of the process (must be unique in the chain)

output\_product: cross reference to the output product of this process, name referres to a product\_XXX section.

output\_timebase: timebase for the processing. Options are “DAILY” or “FILE” see section XXX for a full description of each.

extra\_params: key-values pairs passed in to the process, not normally used

required\_inputXXX: repeated key where XXX is an integer starting at 1 that cross references products in the configuration file that are required for a process to run. There can be any number of required inputs, all must be present for the process to run.

optional\_inputXXX: repeated key where XXX is an integer starting at 1 that cross references products in the configuration file that are optional for a process to run. There can be any number of optional inputs, any may be present for the process to run.

code\_filename: filename of the code that will run as a part of this process

code\_relative\_path: relative path to the code based on mission-rootdir

code\_start\_date: valid range for this code in YYYY-MM-DD format. This code will not be used for dates outside this valid range.

code\_stop\_date: valid range for this code in YYYY-MM-DD format. This code will not be used for dates outside this valid range.

code\_description: human readable decription of the code

code\_version: version of the code. See section XXX for explanation of versions.

code\_output\_interface: The interface version of the created file from this code. See section XXX for explanation of versions.

code\_active: Boolean (True/False) defining if this code is in use.

code\_date\_written: Date the code is written for recordkeeping YYYY-MM-DD

code\_newest\_version: Boolean (True/False) defining if this code is the newest version.

code\_arguments: key=value pairs of values passed along to the code. This is commonly used for the location of configuration files needed by codes.

code\_cpu: number of CPUs used by this code, used to determine how many may run at a time on a given machine.

code\_ram: number of gigabytes of ram used by the code, used to determine how many may run at a time on a given machine.

#### Optional sections

The optional sections and their contents are explained below:

**[DEFUALT]** – This special section populates the key-value pairs into every other section. Useful for key-value pairs that appear many times in the file to save typing.

### 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

Formats are built form combinations of the above, examples:

rbspa\_ect\_hope\_L2\_20130212\_v1.2.3.cdf would be a format of

{SPACECRAFT}\_{PRODUCT}\_{DATE}\_v{VERSION}.cdf where spacecraft-satellite\_name was defined in the config file as “rbspa”, product\_XXX-product\_name was defined as “ect\_hope\_L2”.

20131034\_ns41\_L1.cdf would be {DATE}\_{SPACECRAFT}\_{PRODUCT}.cdf where spacecraft-satellite\_name was defined in the config file as “ns41”, product\_XXX-product\_name was defined as “L2”.

## 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 a Version class
3. An inspector **may** import DBlogging 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['version'] to a instantiated Version object.
8. An inspector **may** set the variable self.diskfile.params['verbose\_provenance'] to any string information to be associated with the file.
9. An inspector **may** set the variable self.diskfile.params['quality\_comment’] to any short comment about the file’s quality.
10. An inspector **may** set the variable self.diskfile.params['caveats'] to any short string containing caveats about the file.
11. An inspector **may** set the variable self.diskfile.params['release\_number'] to any integer related to a release number of the data file.
12. 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.
13. 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.
14. An inspector inspect method **shall** return anything other than None upon filling in the required variables. A string is a recommend return value.

### Example inspector #1

import os

import re

from spacepy import pycdf

from dbprocessing import DBlogging

from dbprocessing import inspector

from dbprocessing import Version

class Inspector(inspector.inspector):

code\_name = 'ect\_L2\_V1.0.0.py'

def inspect(self, kwargs):

re1 = r'{product\_name}\_\d\d\d\d\d\d\d\d.\*.cdf$'.format(\*\*kwargs)

if not re.match(re1, self.basename):

DBlogging.dblogger.debug("Inspector {0}: re did not match {1} {2}".format(self.code\_name, re1, self.basename))

return None

try:

cdf = pycdf.CDF(self.filename)

except:

DBlogging.dblogger.debug("Inspector {0}: error in pycdf.CDF()".format(self.code\_name))

return None # malformed file

try:

self.diskfile.params['utc\_file\_date'] = self.extract\_YYYYMMDD().date()

except:

return None

## get the start time from the file

min\_time = min([v[0] for v in cdf.values() if v.type() in (pycdf.const.CDF\_EPOCH.value, pycdf.const.CDF\_EPOC

H16.value) and v.rv() and len(v) > 0])

max\_time = max([v[-1] for v in cdf.values() if v.type() in (pycdf.const.CDF\_EPOCH.value, pycdf.const.CDF\_EPO

CH16.value) and v.rv() and len(v) > 0])

self.diskfile.params['utc\_start\_time'] = min\_time

self.diskfile.params['utc\_stop\_time'] = max\_time

self.diskfile.params['version'] = inspector.extract\_Version(self.basename)

return "That is not my dog." # anything that is not None is good

### Example Inspector #2

import datetime

import os

import re

from dbprocessing import DBlogging

from dbprocessing import inspector

from dbprocessing import Version

class Inspector(inspector.inspector):

code\_name = 'ephem\_0\_insp.py'

def inspect(self, kwargs):

re1 = r'.\*{product\_name}.\*'.format(\*\*kwargs)

if not re.match(re1, self.basename):

DBlogging.dblogger.debug("Inspector {0}: re did not match {1} {2}".format(self.code\_name, re1, self.bas

ename))

return None

try:

dt = self.extract\_YYYYMMDD()

except:

return None

self.diskfile.params['utc\_file\_date'] = dt.date()

## get the start time from the file

min\_time = dt

max\_time = dt + datetime.timedelta(days=1) - datetime.timedelta(microseconds=1)

self.diskfile.params['utc\_start\_time'] = min\_time

self.diskfile.params['utc\_stop\_time'] = max\_time

try:

self.diskfile.params['version'] = Version.Version(1,0,0)

except:

return None

return "That is not my dog." # anything that is not None is good

## Database structure

DBProcessing maintenance scripts are described in this section and their use and usage. By default all scripts are placed in ~/dbUtils. The used scripts are defined here:

### CreateDB.py

This script creates an empty database ready to be populated from a config file.

CreateDB.py --help

Usage: CreateDB.py [options] filename

Options:

-h, --help show this help message and exit

Typical usage is ~/dbUtils/CreateDB.py mychain.sqlite

### addFromConfig.py

This script reads a config file and sets up a database with the information from the database.

addFromConfig.py --help

Usage: addFromConfig.py [options] -m mission\_db filename

Options:

-h, --help show this help message and exit

-m MISSION, --mission=MISSION

mission to connect to

-v, --verify Don't do anything other than verify the config file

Typical usage is ~/dbUtils/addFromConfig.py –m mychain.sqlite setup.config

### clearProcessingFlag.py

This script clears a processing lock from a locked database that probably crashed.

clearProcessingFlag.py --help

Usage: clearProcessingFlag.py database message

clears the processing flag from a processingthat has crashed

Typical usage is ~/dbUtils/ clearProcessingFlag.py mychain.sqlite “crash fix”

1. http://docs.python.org/2.6/library/configparser.html [↑](#footnote-ref-1)