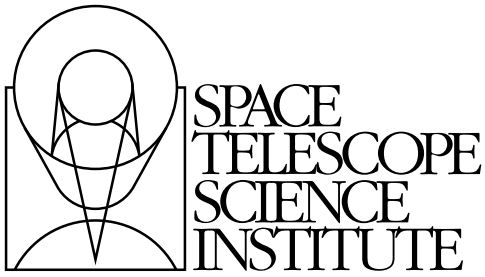


Technical Software Report 1.0a

Calibration and Reference Data System (CRDS)

the crds team
April 27, 2012

CONTENTS

CHAPTER

ONE

WELCOME

Release Notes for CRDS 0.1

Welcome to the Calibration and Reference Data System (CRDS). CRDS manages the reference files and the rules which are used to assign appropriate references to particular data sets. CRDS has a variety of aspects:

- CRDS is a Python package usable within the context of the STSCI Python environment. The core library can be used without network access.
- CRDS has an interactive web server which assists with the creation and submission of "official" reference files and CRDS mappings (rules). CRDS has a database which tracks delivery metadata for references and mappings.
- The CRDS web server also provides network services via JSONRPC related to best reference determination and file distribution.
- CRDS has a Python client package which accesses the network services to determine and locally cache the best references for a dataset.

CHAPTER

TWO

RELEASE NOTES FOR CRDS 0.1

2.1 New Features

2.1.1 General

2.1.2 Under the hood

2.1.3 Changes to Instruments

2.2 Resolved Issues

The following issues have been resolved in this version:

2.2.1 General

2.2.2 Changes to Instruments

2.3 Known Issues

CHAPTER

THREE

INSTALLATION

3.1 Package Overview

From the perspective of an end user, CRDS consists of 3 or more Python packages which implement different capabilities:

- **crds**
  - core package enabling local use and development of mappings and reference files.
- **crds.client**
  - network client library for interacting with the central CRDS server.
- **crds.hst**
  - observatory personality package for HST, with initial mappings for bootstrapping CRDS and defining how HST files are named, located, and certified.
- **crds.jwst**
  - analogous to crds.hst, for JWST, not yet developed.

3.2 Getting the Source Code

At this stage of development, installing CRDS is accomplished by checking CRDS source code out from subversion:

```
% svn co https://subversion.assembla.com/svn/crds/trunk  crds
% cd crds
```

3.3 Setting up your Environment

The CRDS checkout has a template file for the C-shell which defines environment variables, env.csh.

- CRDS_MAPPATH defines the root location where your mappings files will be stored. If left undefined mappings are installed/cached relative to the crds.<observatory> package directory, .e.g. crds.hst.
- CRDS_REFPATH defines the root location where locally cached reference files will be stored. If left undefined references files are cached relative to the crds.<observatory>.references package directory.
- CRDS_SERVER_URL defines the base URL for accessing CRDS network services. For non-developers this will always be set to the current operational CRDS server in env.csh and need not be changed.

Edit env.csh according to your preferences for where to put CRDS files. Then source env.csh to define the variables in your environment:

```
% source env.csh
```

3.4 Run the Install Script

CRDS is partitioned into 3-4 Python packages each of which has it's own setup.py script. To make things easier, the top level directory has a single "install" script which runs all the individual setup.py scripts for you:

```
% ./install
Installing lib
Installing hst
Installing client
Installing django-json-rpc
zip_safe flag not set; analyzing archive contents...
```

3.5 Dependencies

CRDS was developed in and for an STSCI Python environment suitable for pipeline processing. CRDS requires these additional packages to be installed in your Python environment:

- numpy
- pyfits

CHAPTER

FOUR

TOP LEVEL USE

This section describes the formal top level interfaces for CRDS intended as the main entry points for the calibration software or basic use. Functions at this level should be assumed to require network connectivity with the CRDS server.

To function correctly, these API calls require the user to set the environment variables CRDS_REFPATH, CRDS_MAPPATH, and CRDS_SERVER_URL. See the section on *Installation* for more details.

4.1 crds.get_default_context()

Given the observatory name, get_default_context() returns the name of the pipeline mapping which is currently in operational use. The default context defines the matching rules used to determine best reference files for a given set of parameters:

```
def get_default_context(observatory):
    """Return the name of the latest pipeline mapping in use for processing
    files for `observatory`.

    Parameters
    ----------
    observatory  :  name of the observatory the returned context applies to.

        str

        e.g. 'hst' or 'jwst'

    Returns
    -------
    pipeline context mapping

        str

        e.g.   'hst_0007.pmap'
    """
```

4.2 crds.getreferences()

Given dataset header containing parameters required to determine best references, and optionally a specific .pmap to use as the best references context, and optionally a list of the reference types for which reference files to be determined, getreferences() returns a mapping from reference types to reference file base names:

```
def getreferences(header_parameters, reftypes=None, context=None):
    """Return the mapping from the requested `reftypes` to their
    corresponding best reference file names appropriate for a dataset
    described by `parameters` with CRDS rules defined by `context`::

    Parameters
    ----------

    header_parameters :    A mapping of parameter names to parameter value
            strings for parameters which define best reference file matches.

            { str  :   str }

           e.g.  {
                   'INSTRUME' : 'ACS',
                   'CCDAMP' : 'ABCD',
                   'CCDGAIN' : '2.0',
                   ...
                }

    reftypes :    A list of reference type names,  where each reftype is the
                keyword used to record that kind of reference file in a
                dataset header.

                [ str ]

                e.g.  [ 'darkfile', 'biasfile']

                If reftypes is None,  return all reference types defined by
                the instrument mapping for the instrument specified in
                `header_parameters`.

    context :   The name of the pipeline context mapping which should be
            used to define best reference lookup rules,  or None.  If
            `context` is None,  use the latest operational pipeline mapping.

            str

            e.g. 'hst_0037.pmap'

    Returns
    -------
            a mapping from reftypes to best reference file basenames.

            { str : str }

            e.g.   {
                'biasfile' : 'hst_acs_biasfile_0042.fits',
                'darkfile' : 'hst_acs_darkfile_0056.fits',
                }
    """
```

4.3 crds.cache_references()

Given a context filename and a best references mapping returned by getreferences(), cache_references() will download from the CRDS server any reference files not found in the local CRDS reference file cache:

```
def cache_references(context, bestrefs, ignore_cache=False):
    """Cache files specified in `bestrefs` determined by `context` on the
    local machine.   Download any missing files from the CRDS server.

    Parameters
    ----------

    context :     a CRDS pipeline context mapping name

        str

        e.g.   'hst_0037.pmap'

    bestrefs :    a mapping of reftypes to best reference file basenames,
                nominally the return value from getreferences().

            { str : str }

            e.g.   {
                'biasfile' : 'hst_acs_biasfile_0042.fits',
                'darkfile' : 'hst_acs_darkfile_0056.fits',
                }

    ignore_cache :  if True,  download required references even if they
               are already in the CRDS reference file cache.

            bool

    Returns:   A mapping from reference types to file paths for locally
            cached versions of each reference file.

            { str : str }

            e.g.  {
                'biasfile'  :   '/home/jmiller/CRDS/hst/references/hst_acs_biasfile_0042.fits'
                'darkfile'  :   '/home/jmiller/CRDS/hst/reference/hst_acs_darkfile_0056.fits'
            }
    """
```

CHAPTER

FIVE

CLIENT USAGE

This section describes using the crds.client package to access remote web services on the central CRDS server. Remote services support the determination of the best reference files for a dataset. Remote services support transparently downloading CRDS mappings and reference files and caching them locally for use in development or dataset processing. This section is largely superseded by the section on *Top Level Use*, but remains to document lower level features which are network oriented in nature.

5.1 Simplest Useage

After installing CRDS, the simplest usage of CRDS is as follows:

```
% source env.csh
% python
>>> import crds.client as client
>>> bestrefs = client.cache_best_references_for_dataset("hst.pmap", "j8is01j0q_raw.fits")
>>> bestrefs
{'atodtab': '.../site-packages/crds/hst/references/jtab/kcb1734ij_a2d.fits',
 'bpixtab': '.../site-packages/crds/hst/references/jref/m8r09169j_bpx.fits',
 'ccdtab': '.../site-packages/crds/hst/references/jref/o151506bj_ccd.fits',
 'crrejtab': '.../site-packages/crds/hst/references/jref/n4e12510j_crr.fits',
 'darkfile': '.../site-packages/crds/hst/references/jref/n3b10126j_drk.fits',
 'idctab': '.../site-packages/crds/hst/references/jref/p7d1548qj_idc.fits',
 'mdriztab': '.../site-packages/crds/hst/references/jref/ub215378j_mdz.fits',
 'mlintab': '.../site-packages/crds/hst/references/jtab/k9c13374j_lin.fits',
 'oscntab': '.../site-packages/crds/hst/references/jtab/m2j1057pj_osc.fits',
 'pfltfile': '.../site-packages/crds/hst/references/jref/n2d1344mj_pfl.fits',
 'spottab': '.../site-packages/crds/hst/references/jref/r3301467j_csp.fits'}
```

The first parameter is the name of the pipeline context file which should be used to determine best references for the second parameter, a dataset file.

Based on this single function call, the client environment now has all the information and system state required to process the dataset: a mapping from each required reference filekind to a local path to the reference.

Behind the scenes, the following has occurred:

- CRDS used the pipeline context, "hst.pmap", to determine a set of mappings which should be used to compute best references.
- CRDS checked it's local cache for each mapping, and downloaded it from the server if it was not already present locally.
- CRDS asked the server to compute best references based on the relevant parameters extracted from the FITS header. This could also be done locally but asking the server gives an "official" answer.

- CRDS checked it's local cache for each best reference file, and downloaded it from the server if it was not already present locally.

Since I left CRDS_REFPATH undefined for this example, the paths are all relative to .../site-packages/crds/hst/references where ... stands for the absolute path of my Python standard library.

5.2 Just Give Me the Names

cache_best_references_for_dataset() is built upon primitives which do less. A more basic function is to simply determine the names of the best reference files based upon a pipeline context and *header* parameters. Nominally CRDS best reference parameters are extracted from the dataset FITS header, but in principle the *header* can be a dictionary which comes from anywhere:

```
>>> header = { ... what matters ... }
>>> bestrefs = client.get_best_references("hst.pmap", header)
>>> bestrefs
{'atodtab': 'kcb1734ij_a2d.fits',
 'bpixtab': 'm8r09169j_bpx.fits',
 'ccdtab': 'o151506bj_ccd.fits',
 'crrejtab': 'n4e12510j_crr.fits',
 'darkfile': 'n3b10126j_drk.fits',
 'idctab': 'p7d1548qj_idc.fits',
 'mdriztab': 'ub215378j_mdz.fits',
 'mlintab': 'k9c13374j_lin.fits',
 'oscntab': 'm2j1057pj_osc.fits',
 'pfltfile': 'n2d1344mj_pfl.fits',
 'spottab': 'r3301467j_csp.fits'}
```

In this case, get_best_references() started with a dictionary of relevant best reference parameters, *header*, and returned the mapping from reference filekinds to best reference filenames. After executing this function you only know what reference files should be used, not where to get them.

5.3 OK, Now Cache My Files

Once you've decided you're happy with your reference file choices, you can ask CRDS to cache them locally like this:

```
>>> client.cache_references("hst.pmap", bestrefs)
{'atodtab': '.../site-packages/crds/hst/references/jtab/kcb1734ij_a2d.fits',
 'bpixtab': '.../site-packages/crds/hst/references/jref/m8r09169j_bpx.fits',
 'ccdtab': '.../site-packages/crds/hst/references/jref/o151506bj_ccd.fits',
 'crrejtab': '.../site-packages/crds/hst/references/jref/n4e12510j_crr.fits',
 'darkfile': '.../site-packages/crds/hst/references/jref/n3b10126j_drk.fits',
 'idctab': '.../site-packages/crds/hst/references/jref/p7d1548qj_idc.fits',
 'mdriztab': '.../site-packages/crds/hst/references/jref/ub215378j_mdz.fits',
 'mlintab': '.../site-packages/crds/hst/references/jtab/k9c13374j_lin.fits',
 'oscntab': '.../site-packages/crds/hst/references/jtab/m2j1057pj_osc.fits',
 'pfltfile': '.../site-packages/crds/hst/references/jref/n2d1344mj_pfl.fits',
 'spottab': '.../site-packages/crds/hst/references/jref/r3301467j_csp.fits'}
```

5.4 There's an App for That

Maybe you've decided you don't want to worry about figuring out *what matters* and how to format it after all. In that case, call get_minimum_header() on an instrument or reference mapping file and dataset to determine the header parameters that matter:

```
>>> client.get_minimum_header("hst_acs.imap", "test_data/j8bt05njq_raw.fits")
{'CCDAMP': 'C',
 'CCDGAIN': '2.0',
 'DATE-OBS': '2002-04-13',
 'DETECTOR': 'HRC',
 'FILTER1': 'F555W',
 'FILTER2': 'CLEAR2S',
 'FW1OFFST': '0.0',
 'FW2OFFST': '0.0',
 'FWSOFFST': '0.0',
 'LTV1': '19.0',
 'LTV2': '0.0',
 'NAXIS1': '1062.0',
 'NAXIS2': '1044.0',
 'OBSTYPE': 'IMAGING',
 'TIME-OBS': '18:16:35'}
```

The above example uses an instrument context to determine the required parameters to select best references for *all* filekinds.

CHAPTER

SIX

NON-NETWORKED USE

This section describes using the core crds package without access to the network. Using the crds package in isolation it is possible to develop and use new reference files and mappings. Note that a default install of CRDS will also include crds.client and crds.hst or crds.jwst. In particular, the observatory packages define how mappings are named, where they are placed, and how reference files are checked.

6.1 Overview of Features

Using the crds package it's possible to:

- Load and operate on rmaps
- Determine best reference files for a dataset
- Check mapping syntax and verify checksum
- Certify that a mapping and all it's dependencies exist and are valid
- Certify that a reference file meets important constraints
- Add checksums to mappings
- Determine the closure of mappings which reference a particular file.

6.2 Important Modules

There are really two important modules which anyone doing low-level and non- networked CRDS development will first be concerned with:

- **crds.rmap module**
  - **defines classes which load and operate on mapping files**
    - Mapping
    - PipelineContext (.pmap)
    - InstrumentContext (.imap),
    - ReferenceMapping (.rmap)
  - **defines get_cached_mapping() function**
    - loads and caches a Mapping or subclass instances from files, typically this is a recursive process loading pipeline or instrument contexts as well as all associated reference mappings.

    - this *cache* is an object cache to speed up access to mappings, not the file *cache* used by crds.client to avoid repeated network file transfers.

- **crds.selectors module**
    - **defines classes implementing best reference logic**
        - MatchSelector
        - UseAfterSelector
        - Other experimental Selector classes

6.3 Basic Operations on Mappings

6.3.1 Loading Rmaps

Perhaps the most fundamental thing you can do with a CRDS mapping is create an active object version by loading the file:

```
% python
>>> import crds.rmap as rmap
>>> hst = rmap.load_mapping("hst.pmap")
```

The load_mapping() function will take any mapping and instantiate it and all of it's child mappings into various nested Mapping subclasses: PipelineContext, InstrumentContext, or ReferenceMapping.

Loading an rmap implicitly screens it for invalid syntax and requires that the rmap's checksum (sha1sum) be valid by default.

Since HST has on the order of 70 mappings, this is a fairly slow process requiring a couple seconds to execute. In order to speed up repeated access to the same Mapping, there's a mapping cache maintained by the rmap module and accessed like this:

```
>>> hst = rmap.get_cached_mapping("hst.pmap")
```

The behavior of the cached mapping is identical to the "loaded" mapping and subsequent calls are nearly instant.

6.3.2 Seeing Referenced Names

CRDS Mapping classes all know how to show you the files referenced by themselves and their descendents. The ACS instrument context has a reference mapping for each of it's associated file kinds:

```
>>> acs = rmap.get_cached_mapping("hst_acs.imap")
>>> acs.mapping_names()
['hst_acs.imap',
 'hst_acs_idctab.rmap',
 'hst_acs_darkfile.rmap',
 'hst_acs_atodtab.rmap',
 'hst_acs_cfltfile.rmap',
 'hst_acs_spottab.rmap',
 'hst_acs_mlintab.rmap',
 'hst_acs_dgeofile.rmap',
 'hst_acs_bpixtab.rmap',
 'hst_acs_oscntab.rmap',
 'hst_acs_ccdtab.rmap',
 'hst_acs_crrejtab.rmap',
```

```
 'hst_acs_pfltfile.rmap',
 'hst_acs_biasfile.rmap',
 'hst_acs_mdriztab.rmap']
```

The ACS atod reference mapping (rmap) refers to 4 different reference files:

```
>>> acs_atod = rmap.get_cached_mapping("hst_acs_atodtab.rmap")
>>> acs_atod.reference_names()
['j4d1435hj_a2d.fits',
 'kcb1734hj_a2d.fits',
 'kcb1734ij_a2d.fits',
 't3n1116mj_a2d.fits']
```

6.3.3 Computing Best References

The primary function of CRDS is the computation of best reference files based upon a dictionary of dataset metadata. Hence, both an InstrumentContext and a ReferenceMapping can meaningfully return the best references for a dataset based upon a parameter dictionary. It's possible define a header as any Python dictionary provided you have sufficient knowledge of the parameters:

```
>>>  hdr = { ... what matters most ... }
```

On the other hand, if your dataset is a FITS file and you want to do something quick and dirty, you can ask CRDS what dataset metadata may matter for determining best references:

```
>>> hdr = acs.get_minimum_header("test_data/j8bt05njq_raw.fits")
{'CCDAMP': 'C',
 'CCDGAIN': '2.0',
 'DATE-OBS': '2002-04-13',
 'DETECTOR': 'HRC',
 'FILTER1': 'F555W',
 'FILTER2': 'CLEAR2S',
 'FW1OFFST': '0.0',
 'FW2OFFST': '0.0',
 'FWSOFFST': '0.0',
 'LTV1': '19.0',
 'LTV2': '0.0',
 'NAXIS1': '1062.0',
 'NAXIS2': '1044.0',
 'OBSTYPE': 'IMAGING',
 'TIME-OBS': '18:16:35'}
```

Here I say *may matter* because CRDS is currently dumb about specific instrument configurations and is returning metadata about filekinds which may be inappropriate.

Once you have your dataset parameters, you can ask an InstrumentContext for the best references for *all* filekinds for that instrument:

```
>>> acs.get_best_references(hdr)
```

{'atodtab': 'kcb1734ij_a2d.fits', 'biasfile': 'm4r1753rj_bia.fits', 'bpixtab': 'm8r09169j_bpx.fits', 'ccdtab': 'o1515069j_ccd.fits', 'cfltfile': 'NOT FOUND n/a', 'crrejtab': 'n4e12510j_crr.fits', 'darkfile': 'n3o1059hj_drk.fits', 'dgeofile': 'o8u2214mj_dxy.fits', 'flshfile': 'NOT FOUND n/a', 'idctab': 'p7d1548qj_idc.fits', 'imphttab': 'vbb18105j_imp.fits', 'mdriztab': 'ub215378j_mdz.fits', 'mlintab': 'NOT FOUND n/a', 'oscntab': 'm2j1057pj_osc.fits', 'pfltfile': 'o3u1448rj_pfl.fits', 'shadfile': 'kcb1734pj_shd.fits', 'spottab': 'NOT FOUND n/a'}

In the above results, FITS files are the recommended best references, while a value of "NOT FOUND n/a" indicates that no result was expected for the current instrument mode as defined in the header. Other values of "NOT FOUND xxx" include an error message xxx which hints at why no result was found, such as an invalid dataset parameter value or simply a matching failure.

You can ask a ReferenceMapping for the best reference for single the filekind it manages:

```
>>> acs_atod.get_best_ref(hdr)
>>> 'kcb1734ij_a2d.fits'
```

Often it is convenient to simply refer to a pipeline/observatory context file, and hence PipelineContext can also return the best references for a dataset, but this is really just shorthand for returning the best references for the instrument of that dataset:

```
>>> hdr = hst.get_minimum_header("test_data/j8bt05njq_raw.fits")
>>> hst.get_best_references(hdr)
... for this hdr, same as acs.get_best_references(hdr) ...
```

Here it is critical to call get_minimum_header on the pipeline context, hst, because this will make it include the "INSTRUME" parameter needed to choose the ACS instrument.

6.4 Certifying Files

CRDS has a module which will certify that a mapping or reference file is valid, for some limited definition of *valid*. By design only valid files can be submitted to the CRDS server and archive.

6.4.1 Certifying Mappings

For Mappings, crds.certify will ensure that:

- the mapping and it's descendents successfully load
- the mapping checksum is valid
- the mapping does not contain hostile code
- the mapping defines certain generic parameters
- references required by the mapping exist on the local file system

You can check the validity of your mapping or reference file like this:

```
% python -m crds.certify /where/it/really/is/hst_acs_my_masterpiece.rmap
0 errors
0 warnings
0 infos
```

By default, running certify on a mapping *does not* verify that the required reference files are valid, only that they exist.

Later versions of CRDS may have additional semantic checks on the correctness of Mappings but these are not yet implemented and hence fall to the developer to verify in some other fashion.

6.4.2 Certifying Reference Files

For reference files certify has better semantic checks. For reference files, crds.certify currently ensures that:

- the FITS format is valid

- critical reference file header parameters have acceptable values

You can certify reference files the same way as mappings, like this:

```
% python -m crds.certify /where/it/is/my_reference_file.fits
0 errors
0 warnings
0 infos
```

6.5 Mapping Checksums

CRDS mappings contain sha1sum checksums over the entire contents of the mapping, with the exception of the checksum itself. When a CRDS Mapping of any kind is loaded, the checksum is transparently verified to ensure that the Mapping contents are intact.

6.5.1 Ignoring Checksums!

Ordinarily, during pipeline operations, ignoring checksums should not be done. Ironically though, the first thing you may want to do as a developer is ignore the checksum while you load a mapping you've edited:

```
>>> pipeline = rmap.load_mapping("hst.pmap", ignore_checksum=True)
```

6.5.2 Adding Checksums

Once you've finished your masterpiece ReferenceMapping, it can be sealed with a checksum like this:

```
% python -m crds.checksum /where/it/really/is/hst_acs_my_masterpiece.rmap
```

6.6 Which Mappings Use this File?

Particularly in legacy contexts, such as HST, reference file names can be rather cryptic. Further, by design CRDS will have a complex set of fluid and versioned mappings. Hence it may become rather difficult for a human to discern which mappings refer to a particular mapping or reference file. CRDS has the crds.uses module to help answer this question:

```
% python -m crds.uses hst kcb1734ij_a2d.fits
hst.pmap
hst_acs.imap
hst_acs_atodtab.rmap
```

The first parameter indicates the observatory for which files should be considered. Additional parameters specify mapping or reference files which are used. The printed result consists of those mappings which directly or indirectly refer to the used files.

Note that the above results represent the highly simplified context of the current HST prototype, prior to the introduction of mapping evolution and version numbering. In practice, each of the above files might include several numbered versions, and some versions of the above files might not require kcb1734ij_a2d.fits.

crds.uses knows about only the mappings cached locally. Hence the official CRDS server will have a more definitive answer than someone's development machine. The CRDS web site has a link for running crds.uses over all known "official" mappings. crds.uses is especially applicable for understanding the implications of blacklisting a particular file; when a file is blacklisted, all files indicated by crds.uses are also blacklisted.

6.7 Finding Matches for a Reference in a Context

Given a particular context and reference file name, CRDS can also determine all possible matches for the reference within that context:

```
% python -m crds.matches hst.pmap kcb1734ij_a2d.fits
```

```
((('observatory', 'hst'), ('INSTRUME', 'acs'), ('filekind', 'atodtab')), (('DETECTOR', 'HRC'), ('CCDG
((('observatory', 'hst'), ('INSTRUME', 'acs'), ('filekind', 'atodtab')), (('DETECTOR', 'HRC'), ('CCDG
((('observatory', 'hst'), ('INSTRUME', 'acs'), ('filekind', 'atodtab')), (('DETECTOR', 'HRC'), ('CCDG
((('observatory', 'hst'), ('INSTRUME', 'acs'), ('filekind', 'atodtab')), (('DETECTOR', 'HRC'), ('CCDG
((('observatory', 'hst'), ('INSTRUME', 'acs'), ('filekind', 'atodtab')), (('DETECTOR', 'HRC'), ('CCDG
((('observatory', 'hst'), ('INSTRUME', 'acs'), ('filekind', 'atodtab')), (('DETECTOR', 'HRC'), ('CCDG
((('observatory', 'hst'), ('INSTRUME', 'acs'), ('filekind', 'atodtab')), (('DETECTOR', 'HRC'), ('CCDG
((('observatory', 'hst'), ('INSTRUME', 'acs'), ('filekind', 'atodtab')), (('DETECTOR', 'HRC'), ('CCDG
((('observatory', 'hst'), ('INSTRUME', 'acs'), ('filekind', 'atodtab')), (('DETECTOR', 'HRC'), ('CCDG
((('observatory', 'hst'), ('INSTRUME', 'acs'), ('filekind', 'atodtab')), (('DETECTOR', 'HRC'), ('CCDG
((('observatory', 'hst'), ('INSTRUME', 'acs'), ('filekind', 'atodtab')), (('DETECTOR', 'HRC'), ('CCDG
((('observatory', 'hst'), ('INSTRUME', 'acs'), ('filekind', 'atodtab')), (('DETECTOR', 'HRC'), ('CCDG
((('observatory', 'hst'), ('INSTRUME', 'acs'), ('filekind', 'atodtab')), (('DETECTOR', 'HRC'), ('CCDG
((('observatory', 'hst'), ('INSTRUME', 'acs'), ('filekind', 'atodtab')), (('DETECTOR', 'HRC'), ('CCDG
...
```

What is printed out is a sequence of match tuples, with each tuple nominally consisting of three parts:

```
(pmap_imap_rmap_path, match, use_after)
```

Each part in turn consists of nested tuples of the form:

```
(parkey, value)
```

CHAPTER

SEVEN

USING THE CRDS WEB SITE

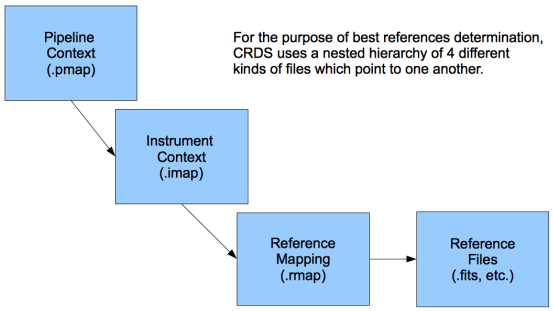
CHAPTER

EIGHT

ABOUT MAPPINGS

CRDS mappings are organized in a 3 tier hierarchy: pipeline (.pmap), instrument (.imap), and reference (.rmap). Based on dataset parameters, the pipeline context is used to select an instrument mapping, the instrument mapping is used to select a reference mapping, and finally the reference mapping is used to select a reference file.

CRDS mappings are written in a subset of Python and given the proper global definitions can be parsed directly by the Python interpreter. Nothing precludes writing a parser for CRDS mappings in some other language.



8.1 Naming

The CRDS HST mapping prototypes which are generated from information scraped from the CDBS web site are named with the forms:

```
<observatory> .pmap                                 .e.g. hst.pmap
<observatory> _ <instrument> .imap                  .e.g. hst_acs.imap
<observatory> _ <instrument> _ <filekind> .rmap     .e.g. hst_acs_darkfile.rmap
```

The names of subsequent derived mappings include a version number:

```
<observatory> _ <version> .pmap                                  .e.g. hst_00001.pmap
<observatory> _ <instrument> _ <version> .imap                   .e.g. hst_acs_00047.imap
<observatory> _ <instrument> _ <version> _ <filekind> .rmap  .e.g. hst_acs_darkfile_00012.rmap
```

8.2 Basic Structure

All mappings have the same basic structure consisting of a "header" section followed by a "selector" section. The header provides meta data describing the mapping, while the selector provides matching rules used to look up the results of the mapping. A critical field in the mapping header is the "parkey" field which names the dataset header parameters which are used by the selector to do its lookup.

8.3 Pipeline Mappings (.pmap)

A sample pipeline mapping for HST looks like:

```
header = {
    'name' : 'hst.pmap',
    'derived_from' : 'created by hand 12-23-2011',
    'mapping' : 'PIPELINE',
    'observatory' : 'HST',
    'parkey' : ('INSTRUME',),
    'description' : 'Initially generated on 12-23-2011',
    'sha1sum' : 'e2c6392fd2731df1e8d933bd990f3fd313a813db',
}

selector = {
    'ACS' : 'hst_acs.imap',
    'COS' : 'hst_cos.imap',
    'NICMOS' : 'hst_nicmos.imap',
    'STIS' : 'hst_stis.imap',
    'WFC3' : 'hst_wfc3.imap',
    'WFPC2' : 'hst_wfpc2.imap',
}
```

A pipeline mapping matches the dataset "INSTRUME" header keyword against its selector to look up an instrument mapping file.

8.4 Instrument Mappings (.imap)

A sample instrument mapping for HST's COS instrument looks like:

```
header = {
    'derived_from' : 'scraped 2011-12-23 11:57:10',
    'description' : 'Initially generated on 2011-12-23 11:57:10',
    'instrument' : 'COS',
    'mapping' : 'INSTRUMENT',
    'name' : 'hst_cos.imap',
    'observatory' : 'HST',
    'parkey' : ('REFTYPE',),
    'sha1sum' : '800fb1567cb5bed4031402c7396aeb86c5e1db61',
    'source_url' : 'http://www.stsci.edu/hst/observatory/cdbs/SIfileInfo/COS/reftablequeryindex',
}

selector = {
    'badttab' : 'hst_cos_badttab.rmap',
    'bpixtab' : 'hst_cos_bpixtab.rmap',
    'brftab' : 'hst_cos_brftab.rmap',
    'brsttab' : 'hst_cos_brsttab.rmap',
```

```
    'deadtab' : 'hst_cos_deadtab.rmap',
    'disptab' : 'hst_cos_disptab.rmap',
    'flatfile' : 'hst_cos_flatfile.rmap',
    'fluxtab' : 'hst_cos_fluxtab.rmap',
    'geofile' : 'hst_cos_geofile.rmap',
    'lamptab' : 'hst_cos_lamptab.rmap',
    'phatab' : 'hst_cos_phatab.rmap',
    'spwcstab' : 'hst_cos_spwcstab.rmap',
    'tdstab' : 'hst_cos_tdstab.rmap',
    'wcptab' : 'hst_cos_wcptab.rmap',
    'xtractab' : 'hst_cos_xtractab.rmap',
}
```

Instrument mappings match the desired reference file type against the reference mapping which can be used to determine a best reference recommendation for a particular dataset. An instrument mapping lists all possible reference types for all modes of the instrument, some of which may not be appropriate for a particular mode. The selector key of an instrument mapping is the value of a reference file header keyword "REFTYPE", and is the name of the dataset header keyword which will record the best reference selection.

8.5 Reference Mappings (.rmap)

A sample reference mapping for HST COS DEADTAB looks like:

```
header = {
    'derived_from' : 'scraped 2011-12-23 11:54:56',
    'description' : 'Initially generated on 2011-12-23 11:54:56',
    'filekind' : 'DEADTAB',
    'instrument' : 'COS',
    'mapping' : 'REFERENCE',
    'name' : 'hst_cos_deadtab.rmap',
    'observatory' : 'HST',
    'parkey' : (('DETECTOR',), ('DATE-OBS', 'TIME-OBS')),
    'sha1sum' : 'e27984a6441d8aaa7cd28ead2267a6be4c3a153b',
}

selector = Match({
    ('FUV',) : UseAfter({
        '1996-10-01 00:00:00' : 's7g1700gl_dead.fits',
    }),
    ('NUV',) : UseAfter({
        '1996-10-01 00:00:00' : 's7g1700ql_dead.fits',
    }),
})
```

Reference mapping selectors are constructed as a nested hierarchy of selection operators which match against various dataset header keywords.

For reference mappings, the header "parkey" field is a tuple of tuples. Each stage of the nested selector consumes the next tuple of header keys. For the example above, the Match operator matches against the value of the dataset keyword "DETECTOR". Based on that match, the selected UseAfter operator matches against the dataset's "DATE-OBS" and "TIME-OBS" keywords to lookup the name of a reference file.

8.6 HST Selectors

For HST, all reference mapping selectors are defined as a two tiered hierarchy with one general matching step (Match) and one date-time match step (UseAfter). All the CRDS selector operators are written to select either a filename *or* a nested operator. In the case of HST, the Match operator locates a nested UseAfter operator which in turn locates the reference file.

8.6.1 Match

Conceptually, the Match operator does a dictionary lookup based on the header keyword values listed in the first tuple of rmap header field *parkey*. In actuality however, CRDS does a winnowing search based on each successive parkey value, eliminating impossible matches and returning the best matching survivor. There are a number of special values associated with the CRDS Match operator: *, Regular Expressions, N/A, and Substitutions.

Wild Cards and Regular Expressions

CRDS supports pseudo-regular expressions in its match patterns. A tuple value of "*" will match any value assigned to that parameter by the dataset. Similarly, literal patterns separated by the "|" punctuator , e.g. "A|B|C|D", will match any one of the patterns, .e.g. "A" or "B" or "C" or "D". These are dubbed "pseudo" regular expressions because "*" is techinically equivalent to "^.*$", and "A|B" is technically equivalent to "^A$|^B$". While losing some flexibility, CRDS simplifies the notation for the sake of brevity and clarity.

N/A

Some of the HST rmaps have match tuple values of "N/A", or Not Applicable. This indicates that the parameter does not "count" for matching that case. A value of N/A is matched as a special version of "*". N/A will match any value in the dataset, or no value, but does not add to the overall sense of goodness of the match. Effectively, N/A parameters are ignored for that match case only, during matching only.

There are a couple uses for N/A parameters by HST. First, sometimes a parameter is irrelevant in the context of the other parameters. Second, sometimes a parameter relevant, but is not stored in the reference file or CDBS, and is not known until supplied by the dataset. In this second case, the unknown parameter can still be used to mutate the values of the other parameters, prior to doing the match. At least initially this technique is used for ACS biasfile with match customization code.

An example of both regular expression and N/A matching occurs in this extract from the ACS biasfile rmap:

```
'parkey' : (('DETECTOR', 'CCDAMP', 'CCDGAIN', 'NUMCOLS', 'NUMROWS', 'LTV1', 'LTV2', 'XCORNER', 'YCORN

('HRC', 'ABCD|AD|BC', '1.0|2.0|4.0|8.0', '1062.0', '1044.0', '19.0', '0.0', 'N/A', 'N/A', 'N/A') : Us
    '1992-01-01 00:00:00' : 'kcb1734jj_bia.fits',
}),
```

This case matches 3 distinct values for CCDAMP, namely ABCD, AD, and BC. It matches 4 distinct values for CCDGAIN, namely 1.0, 2.0, 4.0, and 8.0. The parameters XCORNER, YCORNER, and CCDCHIP are unknown to CDBS and only supplied by actual ACS datasets. They are used to mutate the values of NUMROWS and CCDAMP prior to matching, and hence while they affect the outcome of the match, they are not used literally during the match.

Substitution Parameters

Substituion parameters are short hand notation which eliminate the need to duplicate rmap rules. In order to support WFC3 biasfile conventions, CRDS rmaps permit the definition of meta-match-values which correspond to a set of actual dataset header values. For instance, when an rmap header contains a "substitutions" field like this:

```
'substitutions' : {
    'CCDAMP' : {
        'G280_AMPS' : ('ABCD', 'A', 'B', 'C', 'D', 'AC', 'AD', 'BC', 'BD'),
    },
},
```

then a match tuple line like the following could be written:

```
('UVIS', 'G280_AMPS', '1.5', '1.0', '1.0', 'G280-REF', 'T') : UseAfter({
```

Here the value of G280_AMPS works like this: first, reference files listed under that match tuple define CCDAMP=G280_AMPS. Second, datasets which should use those references define CCDAMP to a particular amplifier configuration, .e.g. ABCD. Hence, the reference file specifies a set of applicable amplifier configurations, while the dataset specifies a particular configuration. CRDS automatically expands substitutions into equivalent sets of match rules.

Match Weighting

Because of the presence of special values like regular expressions, CRDS uses a winnowing match algorithm which works on a parameter-by-parameter basis by discarding match tuples which cannot possibly match. After examining all parameters, CRDS is left with a list of candidate matches.

For each literal, *, or regular expression parameter that matched, CRDS increases its sense of the goodness of the match by 1. For each N/A that was ignored, CRDS doesn't change the weight of the match. The highest ranked match is the one CRDS chooses as best. When more than one match tuple has the same highest rank, we call this an "ambiguous" match. Ambiguous matches will either be merged, or treated as errors/exceptions that cause the match to fail. Talk about ambiguity.

For the initial HST rmaps, there are a number of match cases which overlap, creating the potential for ambiguous matches by actual datasets. For HST, all of the match cases refer to nested UseAfter selectors. A working approach for handling ambiguities here is to merge the two or more equal weighted UseAfter lists into a single combined UseAfter which is then searched.

The ultimate goal of CRDS is to produce clear non-overlapping rules. However, since the initial rmaps are generated from historical mission data in CDBS, there are eccentricities which need to be accomodated by merging or eventually addressed by human beings who will simplify the rules by hand.

8.6.2 UseAfter

The UseAfter selector matches an ordered sequence of date time values to corresponding reference filenames. UseAfter finds the greatest date-time which is less than or equal to (<=) TEXPSTRT of the dataset. Unlike reference file and dataset timestamp values, all CRDS rmaps represent times in the single format shown in the rmap example below:

```
selector = Match({
   ('HRC',) : UseAfter({
       '1991-01-01 00:00:00' : 'j4d1435hj_a2d.fits',
       '1992-01-01 00:00:00' : 'kcb1734ij_a2d.fits',
   }),
   ('WFC',) : UseAfter({
       '1991-01-01 00:00:00' : 'kcb1734hj_a2d.fits',
       '2008-01-01 00:00:00' : 't3n1116mj_a2d.fits',
   }),
})
```

In the above mapping, when the detector is HRC, if the dataset's date/time is before 1991-01-01, there is no match. If the date/time is between 1991-01-01 and 1992-01-01, the reference file 'j4d1435hj_a2d.fits' is matched. If the dataset date/time is 1992-01-01 or after, the recommended reference file is 'kcb1734ij_a2d.fits'.