



crds Documentation

Release 0.1

STScI

May 07, 2013

CONTENTS

1	Installation	1
1.1	Getting the Source Code	1
1.2	Setting up your Environment	1
1.3	Run the Install Script	2
1.4	Test the installation	3
1.5	Package Overview	3
1.6	Dependencies	3
2	Top Level Use	5
2.1	crds.getreferences()	5
2.2	crds.get_default_context()	6
3	Non-Networked Use	7
3.1	Overview of Features	7
3.2	Important Modules	7
3.3	Basic Operations on Mappings	8
3.4	Certifying Files	10
3.5	Mapping Checksums	11
3.6	Which Mappings Use this File?	11
3.7	Finding Matches for a Reference in a Context	12
4	About Mappings	13
4.1	Naming	13
4.2	Basic Structure	14
4.3	Pipeline Mappings (.pmap)	14
4.4	Instrument Mappings (.imap)	14
4.5	Reference Mappings (.rmap)	15
4.6	Selectors	16
5	Command Line Tools	21
5.1	Specifying Files	21
5.2	crds.certify	21
5.3	crds.diff	22
5.4	crds.uses	22
5.5	crds.matches	23
5.6	crds.sync	23
5.7	crds.bestrefs	25
6	Using the CRDS Web Site	29
6.1	Public Functions	29

6.2	Private Functions	35
7	Web Services	43
7.1	Supported Methods	43
7.2	JSONRPC URL	44
7.3	JSONRPC Request	44
7.4	JSONRPC Response	44
7.5	Error Handling	45
7.6	JSONRPC Demo Page	45

INSTALLATION

1.1 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
```

1.2 Setting up your Environment

The CRDS checkout has a template file for the C-shell which defines environment variables, `env.csh`. For JWST development, there are reasonable defaults for everything so you may not need to set these at all.

1.2.1 Basic Environment

- **CRDS_PATH** defines a common directory tree where CRDS reference files and mappings are stored. CRDS_PATH defaults to “/grp/crds/jwst”. Mappings are stored in `${CRDS_PATH}/mappings/[hst|jwst]`. Reference files are stored in `${CRDS_PATH}/references/[hst|jwst]`.
- **CRDS_SERVER_URL** defines the base URL for accessing CRDS network services. CRDS_SERVER_URL defaults to <http://not-a-crds-server.stsci.edu>. Other fallbacks result in operational defaults for JWST only.
- **CRDS_VERBOSITY** enables output of CRDS debug messages. Set to an integer, nominally 50. Higher values output more information, lower values less information.

1.2.2 Typical Build-1 Environments

A typical build-1 environment for CRDS *within STScI* works by aligning the CRDS file cache with the CRDS server’s master copy of reference and mapping files. This makes the master copy of CRDS files on the server appear to be the user’s personal file cache... with the exception that the files are readonly. This approach avoids creating additional copies of the reference files.

HST Settings

```
% setenv CRDS_PATH      /grp/crds/hst
% setenv CRDS_SERVER_URL http://hst-crds.stsci.edu
```

JWST Settings

```
% setenv CRDS_PATH /grp/crds/jwst
% setenv CRDS_SERVER_URL http://not-a-crds-server.stsci.edu
```

Remote Settings

Even for build-1 it's possible to operate CRDS over the network. These settings would create a cache named “crds” in the current working directory when CRDS tools or functions are run:

```
% setenv CRDS_PATH ./crds
% setenv CRDS_SERVER_URL http://jwst-crds.stsci.edu
```

The main caveat is that the CRDS servers are only visible within STScI so outside users must port-forward in order to reach the CRDS server. As an alternative to port forwarding, an on-site user can create a local cache which should continue to work when they are off-site and don't have access to the server or central store.

1.2.3 Advanced Environment

- **CRDS_MAPPATH** can be used to override **CRDS_PATH** and define where only mapping files are stored. If mappings are pre-installed, the directory pointed to by **CRDS_MAPPATH** can be readonly. **CRDS_MAPPATH** defaults to `${CRDS_PATH}/mappings`.
- **CRDS_REFPATH** can be used to override **CRDS_PATH** and define where only reference files are stored. If references are pre-installed, the directory pointed to by **CRDS_REFPATH** can be readonly. **CRDS_REFPATH** defaults to `${CRDS_PATH}/references`.
- **CRDS_CFGPATH** can be used to override **CRDS_PATH** and define where only server configuration information is cached. The directory pointed to by **CRDS_CFGPATH** should be writable. If CRDS is running in server-less mode, this path is irrelevant. **CRDS_CFGPATH** defaults to `${CRDS_PATH}/config`.
- **CRDS_MODE** defines whether CRDS should compute best references using client software (local), server software (remote), or intelligently “fall up” to the server only when the client is deemed obsolete or the server cannot be reached (auto). The default is auto.
- **CRDS_CONTEXT** is an override naming the CRDS pipeline mapping (.pmap) used for computing best references. Ordinarily, CRDS will contact the server to determine the operational pipeline mapping. If the server cannot be reached, CRDS will look in **CRDS_CFGPATH** to determine the last pipeline context the server recommended. If there is no prior server info available in the cache, CRDS will fall-back to using the default pre-installed mappings, e.g. `jwst.pmap`. When **CRDS_CONTEXT** is set, CRDS will ignore server recommendations and availability and use the specified pipeline mapping. However, **CRDS_CONTEXT** will only be used when `context` was specified to `getreferences()` as `None`. If `context` was explicitly specified in a call to `getreferences()` and was not `None`, the specified context will override **CRDS_CONTEXT**. This enables the implementation of command line switches which supercede **CRDS_CONTEXT**.

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
```

1.3 Run the Install Script

CRDS is partitioned into 3-4 Python packages each of which has its 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 client
Installing hst
Installing jwst
Installing tobs
final status 000000
```

1.4 Test the installation

CRDS client testing operates locally and does not require access to the server. Basic CRDS client testing can be done as follows:

```
% ./runtests
... copious test output...
```

```
-----
Ran 59 tests in 13.749s
```

```
OK
```

1.5 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.

1.6 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

For executing the unit tests (runtests) add:

- nose
- BeautifulSoup
- stsci.tools

For building documentation add:

- stsci.sphinxext

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 may require the user to set the environment variables `CRDS_SERVER_URL` and `CRDS_PATH`. See the section on *Installation* for more details.

2.1 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 are to be determined, `getreferences()` will determine best references, cache them on the local file system, and return a mapping from reference types to reference file paths:

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

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

                     { str :    str, int, float, bool }

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

    reftypes :    A list of reference type names.  For HST these are the keywords
                  used to record reference files in dataset headers.  For JWST, these
                  are the identifiers which will appear in instrument contexts and
                  reference mappings.

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

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

ignore_cache : If True, download all required mappings and references
               from the CRDS server. If False, download only those files not
               already in the local caches.

observatory : The name of the observatory this query applies to, needed
              to support both 'hst' and 'jwst' from a single server.

Returns
-----
a mapping from reftypes to cached best reference file paths.

{ str : str }

e.g. {
      'biasfile' : '/path/to/file/hst_acs_biasfile_0042.fits',
      'darkfile' : '/path/to/file/hst_acs_darkfile_0056.fits',
    }
"""
```

2.2 crds.get_default_context()

get_default_context() returns the name of the pipeline mapping which is currently in operational use. When no The default context defines the matching rules used to determine best reference files for a given set of parameters:

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

    Returns
    -----
    pipeline context name

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

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.

3.1 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.

3.2 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

3.3 Basic Operations on Mappings

3.3.1 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 its 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.

3.3.2 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 its 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_crrehtab.rmap',
'hst_acs_pfltfile.rmap',
'hst_acs_biasfile.rmap',
'hst_acs_mdrihtab.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']
```

3.3.3 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 to 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',
'OBSERVE': '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',
'crrehtab': '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.

3.4 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.

3.4.1 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.

3.4.2 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
```

3.5 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.

3.5.1 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)
```

3.5.2 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
```

3.6 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.us 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.us over all known "official" mappings. crds.us is especially applicable for understanding the implications of blacklisting a particular file; when a file is blacklisted, all files indicated by crds.us are also blacklisted.

3.7 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 nst.pmap kcb1734ij_a2d.fits

((( 'observatory', 'hst'), ('INSTRUME', 'acs'), ('filekind', 'atodtab')), (('DETECTOR', 'HRC'), ('CCD', '1')))
((( 'observatory', 'hst'), ('INSTRUME', 'acs'), ('filekind', 'atodtab')), (('DETECTOR', 'HRC'), ('CCD', '2')))
((( 'observatory', 'hst'), ('INSTRUME', 'acs'), ('filekind', 'atodtab')), (('DETECTOR', 'HRC'), ('CCD', '3')))
((( 'observatory', 'hst'), ('INSTRUME', 'acs'), ('filekind', 'atodtab')), (('DETECTOR', 'HRC'), ('CCD', '4')))
((( 'observatory', 'hst'), ('INSTRUME', 'acs'), ('filekind', 'atodtab')), (('DETECTOR', 'HRC'), ('CCD', '5')))
((( 'observatory', 'hst'), ('INSTRUME', 'acs'), ('filekind', 'atodtab')), (('DETECTOR', 'HRC'), ('CCD', '6')))
((( 'observatory', 'hst'), ('INSTRUME', 'acs'), ('filekind', 'atodtab')), (('DETECTOR', 'HRC'), ('CCD', '7')))
((( 'observatory', 'hst'), ('INSTRUME', 'acs'), ('filekind', 'atodtab')), (('DETECTOR', 'HRC'), ('CCD', '8')))
((( 'observatory', 'hst'), ('INSTRUME', 'acs'), ('filekind', 'atodtab')), (('DETECTOR', 'HRC'), ('CCD', '9')))
((( 'observatory', 'hst'), ('INSTRUME', 'acs'), ('filekind', 'atodtab')), (('DETECTOR', 'HRC'), ('CCD', '10')))
((( 'observatory', 'hst'), ('INSTRUME', 'acs'), ('filekind', 'atodtab')), (('DETECTOR', 'HRC'), ('CCD', '11')))
((( 'observatory', 'hst'), ('INSTRUME', 'acs'), ('filekind', 'atodtab')), (('DETECTOR', 'HRC'), ('CCD', '12')))
((( 'observatory', 'hst'), ('INSTRUME', 'acs'), ('filekind', 'atodtab')), (('DETECTOR', 'HRC'), ('CCD', '13')))
((( 'observatory', 'hst'), ('INSTRUME', 'acs'), ('filekind', 'atodtab')), (('DETECTOR', 'HRC'), ('CCD', '14')))
((( 'observatory', 'hst'), ('INSTRUME', 'acs'), ('filekind', 'atodtab')), (('DETECTOR', 'HRC'), ('CCD', '15')))
...

```

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)
```

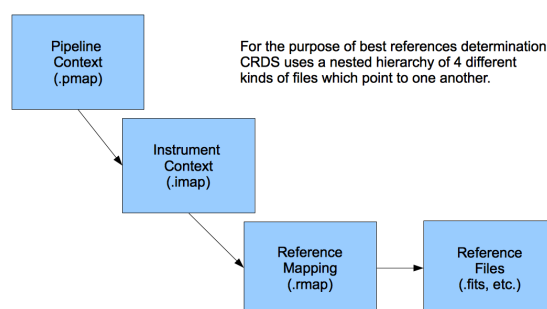
Each part in turn consists of nested tuples of the form:

```
(parkey, value)
```


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.



4.1 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> _ <filekind> _ <version> .rmap .e.g. hst_acs_darkfile_00012.rmap
```

4.2 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.

4.3 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',
    'shasum' : '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.

4.4 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',),
    'shasum' : '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',
    'flxstab' : 'hst_cos_fluxstab.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.

4.5 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')),
    'shasum' : '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.

4.5.1 Parkeys

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.

4.6 Selectors

All the CRDS selection 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.

4.6.1 Match

Based on a dataset's header values, Match locates the match tuple which best matches the dataset. Conceptually this is a dictionary lookup. In actuality, CRDS processes each match parameter in succession, at each step eliminating match candidates that cannot possibly match.

Parameter Tuples and Simple Matches

The CRDS Match operator typically matches a dataset header against a tuple which defines multiple parameter values whose names are specified in the rmap header parkey:

```
("UVIS", "F122LP") : 'some_file_or_nested_selection'
```

Alternately, for simple use cases the Match operator can match against single strings, which is a simplified syntax for a 1-tuple:

```
'UVIS' : 'some_file_or_nested_selection'  
( 'UVIS', ) : 'this_is_the_equivalent_one_tuple'
```

Single Parameter Values

Each value within the match tuples of a Match operator can be an expression in its own right. There are a number of special values associated with each match expression: Ors |, Wildcards *, Regular Expressions (), Literals {}, Relational operators <, >, <=, >=, N/A, and Substitutions.

Or |

Many CRDS match expressions consist of a series of match patterns separated by vertical bars. The vertical bar is read as “or” and means that a match occurs if either pattern matches that dataset header. For example, the expression:

```
("either_this|that", "1|2|3") : "some_file.fits"
```

will match:

```
("either_this", "2")
```

and also:

```
("that", "1")
```

Wild Cards *

By default, * is interpreted in CRDS as a glob pattern, much like UNIX shell file name matching. * matches any sequence of characters. The expression:

```
("F*122",) : "some_file.fits"
```

will match any value starting with “F” and ending with “122”.

Regular Expressions

CRDS can match on true regular expressions. A true regular expression match is triggered by bracketing the match in parentheses ():

```
("(^F[1234]22$)",) : "some_file.fits"
```

The above corresponds to matching the regular expression “`^F[1234]22$`” (note that the bracketing parentheses within the string are removed.) Regular expression syntax is explained in the Python documentation for the `re` module. The above expression will match values starting with “F”, followed by any character which is not “1” or “3” followed by “22”.

Literal Expressions

A literal expression is bracketed with curly braces {} and is matched without any interpretation whatsoever. Hence, special characters like `*` or `|` are interpreted literally rather than as `ors` or wildcards. The expression:

```
("{F|*G}",) : "some_file.fits"
```

matches the value “F*G” as opposed to “F” or anything ending with “G”.

Relational Expressions

Relational expressions are bracketed by the pound character `#`. Relational expressions do numerical comparisons on the header value to determine a match. Relational expressions have implicit variables and support the operators:

```
> >= < <= == and or
```

The expression:

```
("# >1 and <37 #",) : "some_file.fits"
```

will match any number greater than 1 and less than 37.

Between

A special relational operator “between” is used to simply express a range of numbers `>=` to the lower bound and `<` the upper bound, similar to Python slicing:

```
("between 1 47",) : "some_file.fits"
```

will match any number greater than or equal to 1 and less than 47. This is equivalent to:

```
("# >=1 and <47 #",) : "some_file.fits"
```

Note that “between” matches sensibly stack into a complete range. The expressions:

```
("between 1 47",) : "some_file.fits"  
("between 47 90",) : "another_file.fits"
```

provide complete coverage for the range between 1 and 90.

N/A

Some rmaps have match tuple values of “N/A”, or Not Applicable. A value of N/A is matched as a special version of “*”, matching anything, but not affecting the “weight” of the match.

```
('HRC', 'N/A') : “some_file.fits”
```

There are a couple uses for N/A parameters. First, sometimes a parameter is irrelevant in the context of the other parameters. So for an rmap which covers multiple instrument modes, a parameter may not apply to all modes. Second, sometimes a parameter is relevant to custom lookup code, but is not used by the match directly. In this second case, the “N/A” parameter may be used by custom header preconditioning code to assist in mutating the other parameter values that *are* used in the match.

Substitution Parameters

Substitution 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 accommodated by merging or eventually addressed by human beings who will simplify the rules by hand.

4.6.2 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 (\leq) EXPSTART of a 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'

4.6.3 SelectVersion

The SelectVersion() rmap operator uses a software version and various relations to make a selection:

```
selector = SelectVersion({
  '<3.1' : 'cref_flatfield_65.fits',
  '<5' : 'cref_flatfield_73.fits',
  'default' : 'cref_flatfield_123.fits',
})
```

While similar to relational expressions in Match(), SelectVersion() is dedicated, simpler, and more self-documenting. With the exception of default, versions are examined in sorted order.

4.6.4 ClosestTime

The ClosestTime() rmap operator does a lookup on a series of times and selects the closest time which either precedes or follows the given parameter value:

```
selector = ClosestTime({
  '2017-04-24 00:00:00' : "cref_flatfield_123.fits",
  '2018-02-01 00:00:00' : "cref_flatfield_222.fits",
  '2019-04-15 00:00:00' : "cref_flatfield_123.fits",
})
```

So a parameter of '2017-04-25 00:00:00' would select 'cref_flatfield_123.fits'.

4.6.5 GeometricallyNearest

The GeometricallyNearest() selector applies a distance relation between a numerical parameter and the match values. The match value which is closest to the supplied parameter is chosen:

```
selector = GeomtricallyNearest({
  1.2 : "cref_flatfield_120.fits",
  1.5 : "cref_flatfield_124.fits",
})
```

```
    5.0 : "cref_flatfield_137.fits",  
  })
```

In this case, a value of 1.3 would match 'cref_flatfield_120.fits'.

4.6.6 Bracket

The Bracket() selector is unusual because it returns the pair of selections which enclose the supplied parameter value:

```
selector = Bracket({  
    1.2: "cref_flatfield_120.fits",  
    1.5: "cref_flatfield_124.fits",  
    5.0: "cref_flatfield_137.fits",  
  })
```

Here, a parameter value of 1.3 returns the value:

```
('cref_flatfield_120.fits', 'cref_flatfield_124.fits')
```


COMMAND LINE TOOLS

Using the command line tools requires a local installation of the CRDS library. Some of the command line tools also interact with the CRDS server in order to implement their functionality.

5.1 Specifying Files

The command line tools operate on CRDS reference and mapping files in various ways. To specify a file in your local CRDS file cache, as defined by `CRDS_PATH`, use no path on the file:

```
% python -m crds.diff hst.pmap hst_0001.pmap # assumes paths in CRDS cache
```

To specify a particular file which is not located in your cache, give at least a relative path to the file, `./` will do:

```
% python -m crds.diff /some/path/hst.pmap ./hst_0002.pmap # uses given paths
```

5.2 `crds.certify`

`crds.certify` checks a reference or mapping file against constraints on legal matching parameter values. For reference files, `crds.certify` also performs checks of the FITS format and when given a context, and will compare the given file against the file it replaces looking for new or missing table rows.

`crds.certify --help` yields:

```
usage: certify.py [-d] [-e] [-m] [-p] [-t TRAP_EXCEPTIONS] [-x CONTEXT] [-J] [-H]
                files [files ...]
```

Checks a CRDS reference or mapping file.

positional arguments:
files

optional arguments:

<code>-d, --deep</code>	Certify reference files referred to by mappings have valid contents.
<code>-e, --exist</code>	Certify reference files referred to by mappings exist.
<code>-m, --mapping</code>	Ignore extensions, the files being certified are mappings.
<code>-p, --dump-provenance</code>	Dump provenance keywords.
<code>-t TRAP_EXCEPTIONS, --trap-exceptions TRAP_EXCEPTIONS</code>	Capture exceptions at level: pmap, imap, rmap, selector, debug, none
<code>-x CONTEXT, --context CONTEXT</code>	

	Pipeline context defining comparison files.
-J, --jwst	Force observatory to JWST for determining header conventions.
-H, --hst	Force observatory to HST for determining header conventions.

`crds.certify` is invoked as, e.g.:

```
% python -m crds.certify --context=hst_0027.pmap    some_reference.fits
% python -m crds.certify hst.pmap
```

Invoking `crds.certify` on a context mapping recursively certifies all sub-mappings.

5.3 `crds.diff`

`crds.diff` compares two reference or mapping files and reports differences. For references `crds.diff` is currently a thin wrapper around `fitsdiff` but may expand.

For CRDS mappings `crds.diff` performs a recursive logical difference which shows the full match path to each bottom level change. `crds.diff --help` yields:

```
usage: diff.py [-P] [-K] [-J] [-H] old_file new_file
```

Difference CRDS mapping or reference files.

positional arguments:

old_file	Prior file of difference.
new_file	New file of difference.

optional arguments:

-P, --primitive-diffs	Include primitive differences on replaced files.
-K, --check-diffs	Issue warnings about new rules, deletions, or reversions.
-J, --jwst	Force observatory to JWST for determining header conventions.
-H, --hst	Force observatory to HST for determining header conventions.

For standard CRDS filenames, `crds.diff` can guess the observatory. For non-standard names, the observatory needs to be specified. `crds.diff` can be invoked like:

```
% python -m crds.diff    jwst_nircam_dark_0010.fits    jwst_nircam_dark_0011.fits
% python -m crds.diff    jwst_0001.pmap    jwst_0002.pmap
(('hst.pmap', 'hst_0004.pmap'), ('hst_acs.imap', 'hst_acs_0004.imap'), ('hst_acs_darkfile.rmap', 'hst_acs_0004_darkfile.rmap'))
```

5.4 `crds.uses`

`crds.uses` searches the files in the local cache for mappings which refer to the specified files. Since the **local cache** is used only mappings present in the local cache will be included in the results given. `crds.uses` is invoked as:

```
% python -m crds.uses <observatory=hst|jwst> <mapping or reference>...
```

e.g.:

```
% python -m crds.uses hst s7g1700gl_dead.fits
hst.pmap
```

```
hst_cos.imap
hst_cos_deadtab.rmap
```

5.5 crds.matches

`crds.matches` reports the match patterns which are associated with the given reference files:

Usage: `matches.py [options] <context> <references...>`

Options:

```
-h, --help            show this help message and exit
-f, --full            Show the complete match path through the mapping
                      hierarchy.
-V VERBOSITY, --verbose=VERBOSITY
                      Set verbosity level.
```

crds.matches can be invoked as:

```
% python -m crds.matches hst.pmap o8u2214fj_dxy.fits
('HRC', 'CLEAR1S', 'F220W')

% python -m crds.matches --full hst.pmap o8u2214fj_dxy.fits
('hst', 'acs', 'dgeofile', 'HRC', 'CLEAR1S', 'F220W', '2002-03-01', '00:00:00')
```

5.6 crds.sync

The CRDS sync tool is used to download CRDS rules and references from the CRDS server:

usage: `python -m crds.sync`

```
[-h] [--contexts [CONTEXT [CONTEXT ...]]] [--range MIN:MAX] [--all]
[--files [FILES [FILES ...]]] [--datasets [DATASET [DATASET ...]]]
[--fetch-references] [--purge-references] [--purge-mappings] [-i]
[--dry-run] [-v] [--verbosity VERBOSITY] [-V] [-J] [-H]
[--profile PROFILE] [--pdb]
```

Synchronize local mapping and reference caches for the given contexts by downloading missing files from the CRDS server and/or archive.

optional arguments:

```
-h, --help            show this help message and exit
--contexts [CONTEXT [CONTEXT ...]]
                      Specify a list of CRDS mappings to operate on: .pmap, .imap, or .rmap
--range MIN:MAX      Operate for pipeline context ids (.pmaps) between <MIN> and <MAX>.
--all                Operate with respect to all known CRDS contexts.
--files [FILES [FILES ...]]
                      Explicitly list files to be synced.
--datasets [DATASET [DATASET ...]]
                      Cache references for the specified datasets.
--fetch-references    Cache all the references for the specified contexts.
--purge-references    Remove reference files not referred to by contexts from the cache.
--purge-mappings      Remove mapping files not referred to by contexts from the cache.
-i, --ignore-cache    Download sync'ed files even if they're already in the cache.
```

```
--dry-run          Don't remove purged files, just print out their names.
-v, --verbose      Set log verbosity to True, nominal debug level.
--verbosity VERBOSITY
                    Set log verbosity to a specific level: 0..100.
-V, --version      Print the software version and exit.
-J, --jwst         Force observatory to JWST for determining header conventions.
-H, --hst          Force observatory to HST for determining header conventions.
--profile PROFILE  Output profile stats to the specified file.
--pdb              Run under pdb.
```

- Primitive syncing can be done by explicitly listing the files you wish to cache:

```
% python -m crds.sync --files hst_0001.pmap hst_acs_darkfile_0037.fits
```

this will download only those two files.

- Typically syncing CRDS files is done with respect to particular CRDS contexts:

Synced contexts can be explicitly listed:

```
% python -m crds.sync --contexts hst_0001.pmap hst_0002.pmap
```

this will recursively download all the mappings referred to by .pmaps 0001 and 0002.

Synced contexts can be specified as a numerical range:

```
% python -m crds.sync --range 1:3
```

this will also recursively download all the mappings referred to by .pmaps 0001, 002, 0003.

Synced contexts can be specified as `--all` contexts:

```
% python -m crds.sync --all
```

this will recursively download all CRDS mappings for all time.

NOTE: Fetching references required to support contexts has to be done explicitly:

```
% python -m crds.sync --contexts hst_0001.pmap hst_0002.pmap --fetch-references
```

will download all the references mentioned by contexts 0001 and 0002. this can be a huge undertaking and should be done with care.

- Removing files:

Rules/mappings from specified contexts can be removed like this:

```
% python -m crds.sync --contexts hst_0004.pmap hst_0005.pmap --purge-mappings
```

this would remove mapping files which are *not* in 4 or 5.

References from specified contexts can be removed like this:

```
% python -m crds.sync --contexts hst_0004.pmap hst_0005.pmap --purge-references
```

this would remove reference files which are *not* in 4 or 5.

- References for particular datasets can be cached like this:

```
% python -m crds.sync --contexts hst_0001.pmap hst_0002.pmap --datasets <dataset_files..
```

this will fetch all the references required to support the listed datasets for contexts 0001 and
this mode does not update dataset file headers. See also `crds.bestrefs` for header updates.

5.7 crds.bestrefs

`crds.bestrefs` computes the best references with respect to a particular context or contexts for a set of FITS files, dataset ids, or instruments:

```
usage: python -m crds.bestrefs ...
       [-h] [-n NEW_CONTEXT] [-o OLD_CONTEXT] [-c] [-f FILES [FILES ...]]
       [-d IDs [IDs ...]] [-i INSTRUMENTS [INSTRUMENTS ...]]
       [--all-instruments] [-t REFERENCE_TYPES [REFERENCE_TYPES ...]] [-u]
       [--print-affected] [--print-new-references] [-r] [-s] [-v]
       [--verbosity VERBOSITY] [-V] [-J] [-H] [--profile PROFILE] [--pdb]
```

- Determines best references with respect to a context or contexts.
- Optionally compares new results to prior results.
- Optionally prints source data names affected by the new context.
- Optionally updates the headers of file-based data with new recommendations.

Bestrefs has a number of command line parameters which make it operate in different modes:

optional arguments:

```
-h, --help                show this help message and exit
-n NEW_CONTEXT, --new-context NEW_CONTEXT
                        Compute the updated best references using this context. Uses current operation
-o OLD_CONTEXT, --old-context OLD_CONTEXT
                        Compare bestrefs recommendations from two contexts.
-c, --compare-source-bestrefs
                        Compare new bestrefs recommendations to recommendations from data source, f
-f FILES [FILES ...], --files FILES [FILES ...]
                        Dataset files to compute best references for.
-d IDs [IDs ...], --datasets IDs [IDs ...]
                        Dataset ids to compute best references for.
-i INSTRUMENTS [INSTRUMENTS ...], --instruments INSTRUMENTS [INSTRUMENTS ...]
                        Instruments to compute best references for, all historical datasets.
--all-instruments         Compute best references for cataloged datasets for all supported instruments.
-t REFERENCE_TYPES [REFERENCE_TYPES ...], --types REFERENCE_TYPES [REFERENCE_TYPES ...]
                        A list of reference types to process, defaulting to all types.
-u, --update-bestrefs     Update dataset headers with new best reference recommendations.
--print-affected          Print names of data sets for which the new context would assign new reference
--print-new-references    Prints messages detailing each reference file change. If no comparison was
-r, --remote-bestrefs     Compute best references from CRDS server
-s, --sync-references     Fetch the references recommended by new context to the local cache.
-v, --verbose            Set log verbosity to True, nominal debug level.
--verbosity VERBOSITY    Set log verbosity to a specific level: 0..100.
-V, --version            Print the software version and exit.
-J, --jwst              Force observatory to JWST for determining header conventions.
-H, --hst               Force observatory to HST for determining header conventions.
--profile PROFILE        Output profile stats to the specified file.
--pdb                   Run under pdb.
```

5.7.1 New Context

`crds.bestrefs` always computes best references with respect to a context which can be explicitly specified with the `--new-context` parameter. If `--new-context` is not specified, the default operational context is determined by consulting the CRDS server or looking in the local cache.

5.7.2 Lookup Parameter Sources

The two primary modes for `bestrefs` involve the source of reference file matching parameters. Conceptually lookup parameters are always associated with particular datasets and used to identify the references required to process those datasets.

The options `--files`, `--datasets`, `--instruments`, and `--all` determine the source of lookup parameters:

1. To find best references for a list of files do something like this:

```
% python -m crds.bestrefs --new-context hst.pmap --files j8bt05njq_raw.fits j8bt06o6q_raw.fits j
```

the first parameter, `hst.pmap`, is the context with respect to which best references are determined.

2. To find best references for a list of catalog dataset ids do something like this:

```
% python -m crds.bestrefs --new-context hst.pmap --datasets j8bt05njq j8bt06o6q j8bt09jcq
```

3. To do mass scale testing for all cataloged datasets for a particular instrument(s) do:

```
% python -m crds.bestrefs --new-context hst.pmap --instruments acs
```

4. To do mass scale testing for all supported instruments for all cataloged datasets do:

```
% python -m crds.bestrefs --new-context hst.pmap --all
```

or to test for differences between two contexts do:

```
% python -m crds.bestrefs --new-context hst_0002.pmap --old-context hst_0001.pmap --all
```

5.7.3 Comparison Modes

The `--old-context` and `--compare-source-bestrefs` parameters define the best references comparison mode. Each names the origin of a set of prior recommendations and implicitly requests a comparison to the recommendations from the newly computed `bestrefs` determined by `--new-context`.

Context-to-Context

`--old-context` can be used to specify a second context for which `bestrefs` are dynamically computed; `--old-context` implies that a `bestrefs` comparison will be made with `--new-context`. If `--old-context` is not specified, it defaults to `None`.

```
% python -m crds.bestrefs --new-context hst_0042.pmap --old-context hst_0040.pmap \
--instruments acs
```

Prior Source Recommendations

`--compare-source-bestrefs` requests that the bestrefs from `--new-context` be compared to the bestrefs which are recorded with the lookup parameter data, either in the file headers of data files, or in the catalog. In both cases the prior best references are recorded static values, not dynamically computed bestrefs.:

```
% python -m crds.bestrefs --new-context hst_0042.pmap --compare-source-bestrefs \
--datasets j8bt05njq j8bt06o6q
```

5.7.4 Output Modes

`crds.bestrefs` supports several output modes for bestrefs and comparison results to standard out.

If `--print-affected` is specified, `crds.bestrefs` will print out the name of any file for which at least one update for one reference type was recommended. This is essentially a list of files to be reprocessed with new references.:

```
% python -m crds.bestrefs --new-context hst.pmap --files j8bt05njq_raw.fits j8bt06o6q_raw.fits \
--compare-source-bestrefs --print-affected
j8bt05njq_raw.fits
j8bt06o6q_raw.fits
j8bt09jcq_raw.fits
```

5.7.5 Update Modes

`crds.bestrefs` initially supports one mode for updating the best reference recommendations recorded in data files:

```
% python -m crds.bestrefs --new-context hst.pmap --files j8bt05njq_raw.fits j8bt06o6q_raw.fits \
--compare-source-bestrefs --update-bestrefs
```

5.7.6 Verbosity

`crds.bestrefs` has `--verbose` and `--verbosity=N` parameters which can increase the amount of informational and debug output.

USING THE CRDS WEB SITE

CRDS has websites at hst-crds.stsci.edu and jwst-crds.stsci.edu which support the submission, use, and distribution of CRDS reference and mappings files. Functions on the CRDS website are either public functions which do not require authentication or private functions which require a CRDS login account.

HST Calibration and Reference Data System (CRDS)

Open Services

1. [Dataset Best References](#) alpha
2. [Explore Best References](#) alpha
3. [Difference Files](#)
4. [Browse Database](#)
5. [Recent Activity](#)

Restricted Services

1. [Batch Submit References](#)
2. [Edit Fmap](#)
3. [Certify File](#)
4. [Submit References](#)
5. [Submit Mappings](#)
6. [Create Contexts](#)
7. [Set File Enable](#)
8. [Set Default Context](#)

Functions annotated with the word (alpha) are partially completed components of a future build which may prove useful now.

6.1 Public Functions

The following functions are available for anyone with access to the CRDS web server and basically serve to distribute information about CRDS files and recommendations. Initially, the CRDS sites are only visible within the Institute.

6.1.1 Dataset Best References (alpha)

The *Dataset Best References* page supports determining the best references for a single dataset with respect to one CRDS context. Best references are based upon a CRDS context and the parameters of the dataset as determined by the dataset file itself or a database catalog entry.

Context

The context defines the set of CRDS rules used to select best references. *Edit* is the default context from which most newly created contexts are derived. *Operational* is the context currently in use by the pipeline. *Recent* shows the most recently created contexts. *User Specified* enables the submitter to type in the name of any other known context.

Dataset Best References

Context:

▼

hst_0009.pmap 2012-09-30 10:52 (Editing Context)

☒ Editing

hst_0009.pmap 2012-09-30 10:52
or more recent

☐ Operational

hst.pmap 2012-09-29 13:00
or more recent

☐ Recent

hst.pmap 2012-09-29 13:00 ▼

☐ User Specified

e.g. hst_0001.pmap

Dataset:

☒ Upload FITS Header

Choose File No file chosen

☐ Upload Dataset

Choose File No file chosen

☐ Archived Dataset ID

e.g. I9ZF01010

Submit

Dataset

Upload FITS header

Browser-side code can extract the FITS header of a dataset and upload it to the server where best references are computed based on dataset parameters. This function is implemented in Javascript and reliant on HTML5; it supports only parameters present in the FITS primary header. It avoids uploading most of the dataset. It is known to work in Firefox and Chrome but not IE or Safari-5.

Upload Dataset

A user's dataset can be uploaded to the server for best references evaluation.

Archived Dataset

Datasets can be specified by ID and their best reference input parameters will be retrieved from the catalog.

Dataset Best References Results

HST Calibration and Reference Data System (CRDS)



Best References iaai01rtq_raw.fits

Input Parameters

APERTURE: UVS
BINAXIS1: 1.0
BINAXIS2: 1.0
CODAMP: ABCD
CODGAIN: 1.5
CHINJECT: NONE
DATE-OBS: 2009-07-14
DETECTOR: UVS
FILTER: FES5W
INSTRUME: WFC3
REFTYPE: UNDEFINED
SAMP_SEQ: UNDEFINED
SUBARRAY: F
SUBTYPE: UNDEFINED
TIME-OBS: 15:56:09

Recommended Files

ATODTAB: [a50161861_a2d.fits](#) [download](#)
BIASFILE: [v101346r1_bia.fits](#) [download](#)
BPXTAB: [v5420126_bpx.fits](#) [download](#)
CCDTAB: [v291650m1_ccd.fits](#) [download](#)
CRREJTAB: [v014201_crr.fits](#) [download](#)
DARKFILE: [v34201771_drk.fits](#) [download](#)
FLSHFILE: [v701705d1_fs.fits](#) [download](#)
IDCTAB: [v201956r1_idc.fits](#) [download](#)
MDRIZTAB: [v011853a1_mdrz.fits](#) [download](#)
NLINFILE: [n/a](#)
OSCNAB: [v01132101_osc.fits](#) [download](#)
PFLTFILE: [v08161681_pfl.fits](#) [download](#)

Download All: [iaai01rtq_raw_bestrefs.tar.gz](#)

The results page for dataset best references displays the input parameters which were extracted from the dataset header on the right side of the page.

Best reference recommendations are displayed on the left side of the page.

6.1.2 Explore Best References (alpha)

Explore Best References supports entering best references parameters directly rather than extracting them from a dataset or catalog. Explore best references is essentially a sand box which lets someone evaluate what CRDS will do given particular parameter values. The explorer currently lists all parameters which might be relevant to any mode of an instrument and has no knowledge of default values.

The first phase of exploration is to choose a pipeline context and instrument which will be used to define parameter choices:

Explore Best References

Context:

hst_0009.pmap 2012-09-30 10:52 (Editing Context)

Instrument:

*

Submit

The second phase is to enter the parameters of a dataset which are relevant to best references selection.

HST Calibration and Reference Data System (CRDS)

Explore Best References (hst_0023.pmap : cos)

Enter Dataset Parameters

DETECTOR: FUV

EXPTYPE:

LIFE_ADJ: 1

OBSMODE: TIME-TAG

OBSTYPE: SPECTROSCOPIC

OPT_ELEM: 0130M

DATE-OBS: e.g. 2011-01-01

TIME-OBS: e.g. 12:00:00

Get References

Enter Dataset Parameters Select the dataset parameters. We show those parameters that are relevant for the selection of the best reference file associated with any context file. Therefore, some of the parameters will not be relevant to the instrument mode of interest and should be left unselected in order to avoid errors. The results will be determined by the context file and instrument selected.

The entered parameters are evaluated with respect to the given pipeline context and best references are determined. The results are similar or identical to the *Dataset Best References* results.

6.1.3 Difference Files

Difference Files can be used to compare two reference or mapping files. Either the name of a file already in CRDS can be specified (known) or any file can be uploaded via the web (uploaded).

Mapping Differences

For mappings, *Difference Files* displays two kinds of information:

- logical differences where CRDS analyzes the mappings and reports the parameter trail and effect of the difference (add, delete, replace).

HST Calibration and Reference Data System (CRDS)



Difference File

File1:

☒ Known:

☐ Uploaded:

File2:

☒ Known:

☐ Uploaded:

Difference Files compares mapping or reference files to one another using CRDS comparison tools, currently "diff" and "fitsdiff" respectively.

HST Calibration and Reference Data System (CRDS)



Difference hst.pmap against hst_0019.pmap

Mapping Logical Differences

- Logical

Mapping Text Differences

```

+ ('tst_pmap', 'tst_0019.pmap')
+ ('tst_acs.imap', 'tst_acs_0016.imap')
+ ('tst_acs_darkfile.rmap', 'tst_acs_darkfile_0011.rmap')
+ ('tst_cos.imap', 'tst_cos_0002.imap')
+ ('tst_cos_deadhab.rmap', 'tst_cos_deadhab_0003.rmap')
+ ('tst_vfc3.imap', 'tst_vfc3_0001.imap')
+ ('tst_vfc3_biasfile.rmap', 'tst_vfc3_biasfile_0001.rmap')

```

HST Calibration and Reference Data System (CRDS)



Difference hst.pmap against hst_0019.pmap

Mapping Logical Differences

- Logical

[illegible]

- textual differences which show the context difference (diff -c) of the two mapping files.

Mapping Text Differences

```

- (hst.pmap, hst_0019.pmap)

--- hst.pmap      2012-09-08 08:36:20.952512948 -0500
+++ hst_0019.pmap 2012-09-14 14:16:59.820502952 -0500
@@ -1,18 +1,18 @@
header = {
-  'name' : 'hst.pmap',
-  'derived_from' : 'created by hand 12-23-2011',
+  'name' : 'hst_0019.pmap',
+  'derived_from' : 'hst_0018.pmap',
  'mapping' : 'PIPELINE',
  'observatory' : 'HST',
  'parkey' : ('INSTRUME',),
  'description' : 'Initially generated on 2011-11-16 16:23:00',
+  'sha1sum' : 'e2c592fd7310f1e8d932bd99f9fd13a613db',
+  'sha1sum' : '72db93025e2e884835aa7afe3e5d2b5e1f718826',
}

selector = {
-  'ACS' : 'hst_acs.imap',
-  'COS' : 'hst_cos.imap',
+  'ACS' : 'hst_acs_0016.imap',


```

Reference Differences

For references, *Difference Files* is a thin wrapper around the pyfits script *fitsdiff*. Potentially this is useful where a user doesn't have access to pyfits or wants to compare existing reference files without downloading them.

6.1.4 Browse Database

The *Browse Database* feature enables examining the metadata and computable properties of CRDS reference and mapping files.

HST Calibration and Reference Data System (CRDS) 

Browse Database

Filters

Instrument:

Reference Type:

Filename:

User:

Status:

Extension:

Browse Database Searches the CRDS database for files matching the criteria you specify. Filters restrict the search to files matching that value. To ignore a field during search, set it to *.

The first phase is to enter a number of filters to narrow the number or variety of files which are displayed. Leaving any filter at the default value of * renders that constraint irrelevant and all possible files are displayed with respect to that constraint. The result of the first phase is a table of files which matched the filters showing their basic properties.

The second phase is initiated by clicking on the filename link of any file displayed in the table from the first phase. Clicking on a filename link switches to a detailed view of that file only:

The file details page has a number of accordion panes which open when you click on them. All file types have these generic panes:

- Database - lists a table of CRDS metadata for the file.
- Contents - shows the text of a mapping or internal details about a reference file.
- Past Actions - lists website actions which affected this file.
- Used By Files - list known CRDS files which reference this file.

HST Calibration and Reference Data System (CRDS)

Browse Database

Filters

filter	value
instrument	acs

Database Entries

submit date	name	status	submitter	description	instrument	filekind
2012-09-08 08:39	hst_acs_1map	delivered	jmler	initial import	acs	
2012-09-08 08:39	hst_acs_atodtab.map	delivered	jmler	initial import	acs	atodtab
2012-09-08 08:39	hst_acs_bastfile.map	delivered	jmler	initial import	acs	bastfile
2012-09-08 08:39	hst_acs_bpodtab.map	delivered	jmler	initial import	acs	bpodtab
2012-09-08 08:39	hst_acs_ccdtab.map	delivered	jmler	initial import	acs	ccdtab
2012-09-08 08:39	hst_acs_cdtfile.map	delivered	jmler	initial import	acs	cdtfile
2012-09-08 08:39	hst_acs_crestab.map	delivered	jmler	initial import	acs	crestab
2012-09-08 08:39	hst_acs_d2mfile.map	delivered	jmler	initial import	acs	d2mfile
2012-09-08 08:39	hst_acs_darkfile.map	delivered	jmler	initial import	acs	darkfile
2012-09-08 08:39	hst_acs_dgeofile.map	delivered	jmler	initial import	acs	dgeofile
2012-09-08 08:39	hst_acs_dkcfle.map	delivered	jmler	initial import	acs	dkcfle
2012-09-08 08:39	hst_acs_fishfile.map	delivered	jmler	initial import	acs	fishfile
2012-09-08 08:39	hst_acs_idctab.map	delivered	jmler	initial import	acs	idctab
2012-09-08 08:39	hst_acs_imphtab.map	delivered	jmler	initial import	acs	imphtab
2012-09-08 08:39	hst_acs_mdctab.map	delivered	jmler	initial import	acs	mdctab

HST Calibration and Reference Data System (CRDS)

Browse [hst_acs_darkfile.rmap](#) [edit](#) [download](#)

Database
Contents
Past Actions
Used By Files

Reference files have these additional panes:

- Certify Results - shows the results of `crds.certify` run on this reference now.
- Lookup Patterns - lists the parameters sets which lead to this reference.

6.1.5 Recent Activity

The *Recent Activity* view shows a table of the actions on CRDS files which are tracked. Only actions which change the states of files in some way are tracked:

HST Calibration and Reference Data System (CRDS)

List Actions

Recent Activity Search the CRDS database for actions matching the filters you specify. To ignore a field during search, set it to *.

Filters

Action:

Instrument:

Reference Type:

Filename:


User:

Status:

Extension:

The first page lists a number of constraints which can be used to choose activities of interest. To ignore any constraint, leave it set at the default value of *. The result of the activity search is a table of matching actions:

The details vary by the type of action, in this case showing the original name of a file prior to submission to CRDS and the assignment of its official name.



HST Calibration and Reference Data System (CRDS)

Recent Activity

Filters

filter	value
action	submit file
instrument	acs

Activity

date	filename	action	user	why	details
2012-09-10 20:05	tbl_acs_dadfile_0007.fits	submit file	jmler	Something.	{["v441.43468_dsk.fits", "tbl_acs_dadfile_0007.fits"], []}

6.2 Private Functions

The following functions are restricted to users with accounts on the CRDS website and support the submission of new reference and mapping files and maintenance of the overall site.

6.2.1 Batch Submit References

Batch Submit References is intended to handle the majority of CRDS reference submissions with a high degree of automation. This page accepts a number of reference files and metadata which is applied to all of them. The specified reference files are checked on the server using `crds.certify` and if they pass are submitted to CRDS. All of the submitted references must be of the same reference type, i.e. controlled by the same `.rmap` file. Tabular reference files are checked with respect to the derivation context by `crds.certify`.

HST Calibration and Reference Data System (CRDS)	
Batch Submit Reference	
Upload Files	Batch Submit References supports uploading a list of reference files for a single instrument and reference type, i.e. one rmap. After checking and storing the reference files, new CRDS mapping files which refer to them are generated.
<input type="button" value="Upload Files"/>	Upload Files lets you upload new references to CRDS. Each reference file must certify or the entire batch will be rejected.
Derive From Context	Derive From Context The name of the pipeline context which will be recursively updated to insert the new reference file and create a new context which includes them.
<input type="button" value="Derive From Context"/>	Change Level The degree to which the new files are expected to impact science results.
Change Level: <input type="text" value="SEVERE"/>	Creator author of the contents or changes to the file, not necessarily the submitter. The submitter is known based on login information.
Creator: <input type="text"/>	Description Information about what's new in this file and what the expected impacts are.
Description: <input type="text"/>	Auto Rename when checked, the uploaded file will be renamed to a unique name using CRDS naming conventions and id numbers, e.g. "tbl_700grm_dadfile_0007.fits". Left unchecked, this option enables easy comparisons with COBS names during side-by-side testing with CRDS. Once CRDS is operational Auto Rename should be left unchecked so that CRDS-style names are assigned.
Auto-Rename: <input type="checkbox"/>	Compare Old Reference check this to perform table comparisons against prior versions of the submitted reference file identified by the comparison context. If left unchecked, the comparison context is ignored and no comparison to prior versions is performed.
Compare Old Reference: <input type="checkbox"/>	
<input type="button" value="Submit References"/>	

Upload Files

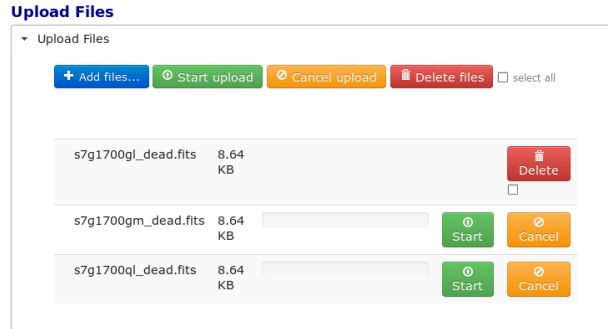
The first task involved with *Batch Submit References* is transferring the submitted files to the server. For CRDS build-2, there are two approaches for getting files on the server, web based and shell based. Both approaches involve transferring files to an ingest directory in the CRDS filestore. Each CRDS user will have their own ingest directory. Initially the only user is "test". This section applies equally to all of the file submission pages that have an *Upload Files* accordion.

Web Approach

On the file submission pages, the *Upload Files* accordion opens to support uploading submitted files to a user's CRDS ingest directory via the browser.

Uploading files is accomplished by:

- Opening the accordion panel by clicking on it.



- Add files to the upload list by clicking on the *Add Files...* button. Alternately for modern browsers (Chrome) drag-and-drop files from your desktop to the upload accordion.
- Click *Start Upload* to initiate the file transfer. You should see a progress bar(s) showing the status of the upload(s). When the upload successfully completes the buttons will change to *delete*.
- Click *Delete* for any file added by mistake or for failed uploads.
- Click *Cancel Upload* to abort a file transfer during the upload.
- Close the accordion panel by clicking on it.

IMPORTANT Just adding files to the file list does not upload them. You must click *Start upload* to initiate the file transfer. In the screenshot above, the file with the *delete* button next to it is already on the server in the ingest directory. The files with *start* and *cancel* buttons next to them have only been declared as candidates for upload. To finish uploading all 3 files, check *select all* and click *Start upload*.

Shell Approach

In the shell approach a user must login to UNIX (in some fashion) and transfer files into their CRDS ingest directory manually. The nominal approach for doing this is to use the cp or scp commands. For instance, from my home, having already set up ssh and scp access, I might say:

```
% scp /this_delivery/*.fits thor.stsci.edu:/grp/crds/hst/ingest/test
```

Abstractly this is:

```
% scp <submitted reference files...> <host>:/grp/crds/<observatory>/ingest/<crds_username>
```

The submitted reference files should now be in the ingest directory for *HST* user *test*. Once the files are in the ingest directory, the CRDS web server will behave as if they had been uploaded through web interface. Refreshing the file submission web page should make manually copied files show up in the *Upload Files* accordion.

The purpose of using cp or scp is to improve the efficiency and reliability of the file transfers. Files transferred to the ingest directory via shell should still be removeable using the *Upload Files* delete buttons.

Cleaning Up

No matter which file transfer approach you use, transferring many large references to the CRDS ingest directories can take a long time. As long as the files remain in the ingest directory, it is possible to submit them more than once (if things go wrong) without uploading again. Consequently, for build-2, file removal from the ingest directory is left as a user directed activity. To remove files from your ingest directory, either use “rm” in the shell, or use the delete buttons in *Upload Files*.

Derive From Context

The specified context is used as the starting point for new automatically generated context files and also determines any predecessors of the submitted references for comparison during certification. If all the submitted reference files pass certification, a new .rmap, .imap, and .pmap are generated automatically to refer to the newly entered references. Based on their header parameters, references are automatically assigned to appropriate match locations in the .rmap file.

Derive From Context

▼ hst_0002.pmap 2012-09-30 09:33 (Editing Context)

☒ Editing hst_0002.pmap 2012-09-30 09:33
or more recent

☐ Operational hst.pmap 2012-09-29 13:00
or more recent

☐ Recent hst.pmap 2012-09-29 13:00 ▼

☐ User Specified e.g. hst_0001.pmap

There are two special contexts in CRDS which are tracked:

Edit Context

Edit Context is the default context used for editing. Whenever a new .pmap is created or added, it becomes the editing context from which other .pmaps are derived by default.

Operational Context

Operational Context is the .pmap which is nominally in use by the pipeline. Generally speaking, multiple contexts might be added to CRDS as the Edit Context long before they become operational.

Recent

Recent lists a number of recently added contexts based on delivery time.

User Specified

Any valid CRDS context can be typed in directly as User Specified.

Auto Rename

Normally files uploaded to CRDS will be assigned new unique names. During side-by-side testing with CDBS, *Auto Rename* can be deselected so that new files added to CRDS retain their CDBS names for easier comparison. The CRDS database remembers both the name of the file the submitter uploaded as well as the new unique name.

Compare Old Reference

When checked CRDS will certify incoming tabular references against the files they replace with respect to the derivation context. For other references this input is irrelevant and ignored.

Results

The results page lists the following items:

- *Starting Context* is the context this submission derive from.

Batch Reference Submit Results**Starting Context**

hst_0002.pmap

Generated New Mappings

hst_0003.pmap

hst_cos_0003.imap

hst_cos_deadtab_0003.rmap

Certify Results

```
› s7g1700ql_dead.fits --> hst_cos_deadtab_0004.fits OK
› s7g1700gl_dead.fits --> hst_cos_deadtab_0005.fits OK
› s7g1700gm_dead.fits --> hst_cos_deadtab_0006.fits OK
```

Actions on hst_cos_deadtab_0002.rmap

```
› Rmap Logical Diffs
› Rmap Text Diffs
```

Confirm or Abort Submission

- *Generated New Mappings* lists the new mapping files which provide the generated context for using the submitted references.
- *Actions on Rmap* provides two accordions showing how the rmap controlling the submitted references was modified. The logical differences accordion has a table of actions, either *insert* for completely new files or *replace* for files which replaced an existing file. The text differences are essentially output from UNIX *diff* for the old and new rmaps.
- *Certify Results* has an accordion panel for each submitted reference file which contains the results from crds.certify. The submitted name of each file is listed first, followed by any official name of the file assigned by CRDS. The status of the certification can be “OK” or “Warnings”. Warnings should be reviewed by opening the accorion panel.

IMPORTANT The results page only indicates the files which will be added to CRDS if the submission is *confirmed*. Prior to confirmation of the submission, neither the submitted references nor the generated mappings are officially in CRDS. Do not *leave the confirmation page* prior to confirming.

Collisions

Under some circumstances, a *Collision Warning* accordion will be present. It should be carefully examined to ensure that overlapping edits of the same context file have not occurred. Overlaps can be resolved by cancelling the current submission and re-doing it, or by accepting the current submission and manually correcting the mappings involved. Failure to correctly resolve a collision will most likely result in one of two sets of conflicting changes being lost.

✖ Edit Collision Warnings

[hst_cos_deadtab_0012.fits](#) and [hst_cos_deadtab_0001.fits](#) were both derived from [s7g1700ql_dead.fits](#)
[hst_cos_deadtab_0012.fits](#) and [hst_cos_deadtab_0004.fits](#) were both derived from [s7g1700ql_dead.fits](#)

This file was derived from the same parent file as one or more others.
This suggests that two people edited the same file at the same time and will not incorporate one another's changes.
The person receiving this warning should derive a new update, most likely from the best of the files this submission collided with.

Do not confirm the file(s) submitted/generated here unless the collision is known to be harmless.

Collision tracking for CRDS mappings files is done based upon header fields, nominally the *name* and *derived_from* fields. These fields are automatically updated when mappings are submitted or generated.

Collision tracking for reference files is currently filename based. The submitted name of a reference file is assumed to be the same as the file it was derived from. This fits a work-flow where a reference is first downloaded from CRDS, modified under the same name, and re-uploaded. Nominally, submitted files are automatically re-named.

Confirm or Discard

If everything looks good the last step is to click the *Confirm* button. Clicking the Confirm button finalizes the submission process, submits the files for archive pickup, and makes them a permanent part of CRDS visible in the database browser and potentially redistributable. A confirmed submission cannot be revoked, but neither will it go into use until the pipeline or a user explicitly requests it.

Discarding a batch submission based on warnings or bad rmap modifications removes the submission from CRDS. In particular temporary database records and file copies are removed.

Following any CRDS pipeline mapping submission, the default *edit* context is updated to that pipeline mapping making it the default starting point for future submissions.

6.2.2 Certify File

Certify File runs `crds.certify` on the files in the ingest directory.

If the certified file is a reference table, the specified context is used to locate a comparison file.

6.2.3 Submit References

Submit References provides a lower level interface for submitting a list of references which don't have to be of the same instrument and filetype. No context mappings are generated to refer to the submitted files. Submitted references must still pass through `crds.certify`.

6.2.4 Submit Mappings

Submit Mappings provides a basic interface for submitting a list of mapping files which don't have to be related. This can be used to submit context files which refer to files from *Submit References* and with fewer restrictions on allowable changes. Typically only `.rmaps` are submitted this way. Mappings submitted this way must also pass through `crds.certify`.

Submit Reference Files

Upload Files

» Upload Files

Comparison Context

» hst_0009.pmap 2012-09-30 10:52 (Editing Context)

Compare Old Reference: ☐

Change Level: SEVERE

Creator:

Description:

Auto-Rename: ☐

Submit Files

Submit Mapping Files

Upload Files

» Upload Files

Change Level: SEVERE

Creator:

Description:

Auto-Rename: ☐

Submit Files

6.2.5 Create Contexts

Create Contexts provides a basic interface for automatically generating pipeline and instrument context mappings which refer to the specified reference mapping files.

Using *Create Contexts* the upper level mappings can be modified to refer to a number of (most likely hand-edited) reference mappings. Rmaps referred to by create contexts must already be known to CRDS. *Create Contexts*

6.2.6 Set File Enable

Set File Enable provides control over the Blacklist and Reject attributes of a file.

Rejecting a file is used to signal that the file should no longer be used. Rejecting a file affects only that file. Blacklisting a file marks the file as unusable, but it also blacklists all files which directly or indirectly refer to the original blacklisted file. So, blacklisting is transitive, but rejection is intransitive. Either blacklisting or rejection can be undone by marking the file as OK again using *Set File Enable*. Only files which are already known to CRDS can be rejected or blacklisted.

Create Parent Contexts

Derived From

hst_0009.pmap 2012-09-30 10:52 (Editing Context)

New Rmaps

```
hst_acs_darkfile_0027.rmap
hst_acs_biasfile_0042.rmap
```

Description

Why these rmaps were hand modified.

Create Parent Contexts

HST Calibration and Reference Data System (CRDS)



Set File Enable

File:

Reject Type:

Mark as:

Reason:

Blacklist File Marks a file as "bad" or "ok". Attempts to use a "bad" file for best reference determinations will generate an error.

File a known CRDS mapping or reference filename to be blacklisted.

Mark as Choose "bad" to disable a file. Choose "ok" to reenable.

Reason Why this file is no longer usable or is usable once again.

WEB SERVICES

The CRDS servers support a JSONRPC based service mechanism which enables remote users to make calls to the CRDS server without installing the CRDS Python based client library. See <http://json-rpc.org/wiki/specification> for more details on the JSONRPC protocol.

7.1 Supported Methods

7.1.1 `get_default_context(observatory)`

get_default_context returns the name of the pipeline mapping which is currently in use by default in the operational pipeline, e.g. 'jwst_0001.pmap'. `get_default_context` is called with a single parameter, *observatory*, which can be 'hst' or 'jwst'.

7.1.2 `get_best_references(context, header, reftypes)`

get_best_references matches a set of parameters *header* against the lookup rules specified by the pipeline mapping *context* to return a mapping of type names onto recommended reference file names.

A suitable *context* string can be obtained from `get_default_context()` above, although any archived CRDS context file can be specified.

The *header* parameter of `get_best_references` is nominally a JSON object which maps CRDS parkey names onto dataset file header values. CRDS parkey names can be located by browsing reference mappings (.rmap's) and looking at the *parkey* header parameter of the rmap.

For JWST, the rmap parkeys (matching parameter names) are currently specified as JWST stpipe data model dotted identifiers. Example JSON for the `get_best_references` *header* parameter for JWST is:

```
{ "meta.instrument.type": "fgs",  
  "meta.instrument.detector": "fgs1",  
  "meta.instrument.filter": "any" }
```

For JWST, it is also possible to use the equivalent FITS header keyword, as defined by the data model schema, to determine best references:

```
{ "instrume": "fgs",  
  "detector": "fgs1",  
  "filter": "any" }
```

For HST, GEIS or FITS header keyword names are supported.

reftypes should be a json array of strings, each naming a single desired reference type. If *reftypes* is passed as null, recommended references for all reference types are returned. Reference types which are defined for an instrument but which are not applicable to the mode defined by *header* are returned with the value *NOT FOUND n/a*.

Example JSON for *reftypes* might be:

```
["amplifier", "mask"]
```

7.2 JSONRPC URL

The base URL used for making CRDS JSONRPC method calls is essentially */json/*. All further information, including the method name and the parameters, are POSTed using a JSON serialization scheme. Example absolute server URLs are:

7.2.1 JWST

```
http://jwst-crds.stsci.edu/json/
```

7.2.2 HST

```
http://hst-crds.stsci.edu/json/
```

7.3 JSONRPC Request

An example CRDS service request can be demonstrated in a language agnostic way using the UNIX command line utility *curl*:

```
% curl -i -X POST -d '{"jsonrpc": "1.0", "method": "get_default_context", "params": ["jwst"], "id": 1}'
HTTP/1.1 200 OK
Date: Fri, 12 Oct 2012 17:29:46 GMT
Server: Apache/2.2.3 (Red Hat) mod_python/3.3.1 Python/2.7.2
Vary: Cookie
Content-Type: application/json-rpc
Connection: close
Transfer-Encoding: chunked
```

The *jsonrpc* attribute is used to specify the version of the JSONRPC standard being used, currently 1.0 for CRDS.

The *method* attribute specifies the name of the service being called.

The *params* attribute specifies a JSON array of parameters which are passed positionally to the CRDS method.

The *id* can be used to associate calls with their responses in asynchronous environments.

7.4 JSONRPC Response

The response returned by the server for the above request is the following JSON:

```
{"error": null, "jsonrpc": "1.0", "id": 1, "result": "jwst_0000.pmap"}
```


7.5 Error Handling

Because `get_best_references` determines references for a list of types, lookup errors are reported by setting the value of a reference type to “NOT FOUND ” + `error_message`. A value of “NOT FOUND n/a” indicates that CRDS determined that a particular reference type does not apply to the given parameter set.

Fatal errors are handled by setting the error attribute of the result object to an error object. Inspect the `result.error.message` attribute to get descriptive text about the error.

7.6 JSONRPC Demo Page

The CRDS servers support demoing the JSONRPC services and calling them interactively by visiting the URL `.../json/browse/`. The resulting page is shown here:

An example dialog for `get_best_references` from the CRDS jsonrpc demo page is shown here with FITS parkey names:

```
>>> jsonrpc.get_best_references("jwst_0000.pmap", {'INSTRUME':'FGS','DETECTOR':'FGS1', 'FILTER':'ANY'})
Requesting ->
{"id":"jsonrpc", "params":["jwst_0000.pmap", {"INSTRUME":"FGS", "DETECTOR":"FGS1", "FILTER":"ANY"}], n
Deferred(12, unfired)
Got ->
{"error": null, "jsonrpc": "1.0", "id": "jsonrpc", "result": {"linearity": "jwst_fgs_linearity_0000.l
```

And the same query is here with JWST data model parkey names:

```
>>> jsonrpc.get_best_references("jwst_0000.pmap", {'META.INSTRUMENT.TYPE':'FGS','META.INSTRUMENT.DETE
Requesting ->
{"id":"jsonrpc", "params":["jwst_0000.pmap", {"META.INSTRUMENT.TYPE":"FGS", "META.INSTRUMENT.DETECTOR
Deferred(14, unfired)
Got ->
{"error": null, "jsonrpc": "1.0", "id": "jsonrpc", "result": {"linearity": "jwst_fgs_linearity_0000.l
```

NOTE: An apparent bug in the demo interpreter makes it impossible to pass the `get_best_references` *reftypes* parameter as an array of strings. In the current demo *reftypes* can only be specified as null.

JSON-RPC Browser

Methods

get_reference_names

get_best_references

get_file_chunk

get_url

list_mappings

get_server_info

file_exists

get_file_info

get_default_context

get_mapping_names

Console

```
{ "error": { "executable": "/usr/local/bin/python2.7", "code": 500, "name":
"OtherError", "message": "OtherError: Missing 'META.INSTRUMENT.TYPE' keyword in header",
"data": null, "stack": "Traceback (most recent call last):\n File \"/grp/crds/jwst/webserver
/python/lib/python/django_json_rpc-0.6.2-py2.7.egg/jsonrpc/site.py", line 152, in
response_dict\n R = apply_version[version](method, request, D['params'])\n File \"/grp
/crds/jwst/webserver/python/lib/python/django_json_rpc-0.6.2-py2.7.egg/jsonrpc/site.py", line
125, in <lambda>\n '1.0': lambda f, r, p: f(r, *p)}\n File \"/grp/crds/jwst/webserver/python
/lib/python/crds/server/jsonapi/views.py", line 141, in get_best_references\n return
rmap.get_best_references(context, conditioned, include=reftypes)\n File \"/grp/crds/jwst
/webserver/python/lib/python/crds/rmap.py", line 1057, in get_best_references\n return
ctx.get_best_references(header, include=include)\n File \"/grp/crds/jwst/webserver/python
/lib/python/crds/rmap.py", line 501, in get_best_references\n instrument =
self.get_instrument(header)\n File \"/grp/crds/jwst/webserver/python/lib/python/crds/rmap.py",
line 583, in get_instrument\n raise crds.CrdsError("\nMissing '%s' keyword in header\n" %
self.instrument_key)\nCrdsError: Missing 'META.INSTRUMENT.TYPE' keyword in header\n"},
"jsonrpc": "1.0", "id": null, "result": null}
>>> jsonrpc.get_best_references("jwst_0000.pmap",
{'META.INSTRUMENT.TYPE': 'FGS', 'META.INSTRUMENT.DETECTOR': 'FGS1',
'META.INSTRUMENT.FILTER': 'ANY'}, null)
Requesting ->
{"id": "jsonrpc", "params": [{"jwst_0000.pmap", {"META.INSTRUMENT.TYPE": "FGS",
"META.INSTRUMENT.DETECTOR": "FGS1", "META.INSTRUMENT.FILTER": "ANY"}, null],
"method": "get_best_references", "jsonrpc": "1.0"}
Deferred(4, unfired)
Got ->
{"error": null, "jsonrpc": "1.0", "id": "jsonrpc", "result": {"linearity":
"jwst_fgs_linearity_0000.fits", "amplifier": "jwst_fgs_amplifier_0000.fits", "mask":
"jwst_fgs_mask_0000.fits"}}
```

- JSON-RPC methods are exposed through the `jsonrpc` global.
- Call a method just as you would a javascript one: `jsonrpc.jsonrpc.test('arg1')`
- Calling a method returns a `Deferred` (see the [MochiKit docs](#)). As soon as the deferred returns the result is available by the global variables `json_result` or `json_error`
- Use `dir` to explore the namespace: `dir(jsonrpc)`