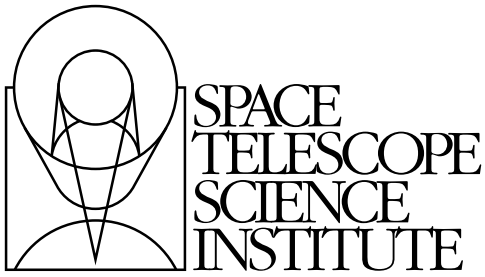

Technical Software Report 2010-02

Analysis of Calibration Reference Data System Design and Requirements: DRAFT

Perry Greenfield
June 15, 2010

Contents

Abstract

This document contains an analysis of what requirements are practical for a calibration reference data system.

1 Introduction

I've been giving the "new" (i.e., what is to be used for JWST) CDBS some thought. So here are some thoughts.

To distinguish the JWST CDBS system, from now on I will call it the Calibration Reference Data System (CRDS). [This is open to change of course]

2 Purpose

I think it is important to keep in mind what the purposes are, and the priority of the purposes (and features). I'll summarize my view of that. I could well be missing important aspects or features (one reason I'm writing this is to solicit corrections!)

1. Identifying the most appropriate calibration reference data for a given dataset. This seems to me to be–far and away–the most important purpose.
2. Keeping track of when and why calibration reference data have been entered into the system. This involves keeping some metadata about the calibration reference files so the appropriate history can be recorded.
3. Ensuring that calibration reference data are appropriately archived.
4. Being able to answer various questions regarding calibration reference files. This item is a bit murky, and it isn't clear to me how much this aspect has really been used for the existing CDBS system, e.g., how often user-level queries are performed (as opposed to queries performed to accomplish items 1-3).

3 New capabilities needed

Currently, CDBS only is used internally. That is to say, that the current pipeline system obtains the best reference files using code not easily used by programs running on the outside. We would like to see CRDS work differently in that its services be available to anyone with a network connection. It must be usable as a web service, and for the purposes of

ensuring consistency, we would like to see our operational pipelines use it exactly the same way that an external user running the calibration pipelines on their local computers would, at least with the identical software interface (perhaps the access may be different to enable faster processing at STScI)

Item 1 is not yet very clearly specified. There are aspects of that and making it available through the web that introduce some complexity that we have not had to deal with in the current CDBS and for which the current CDBS cannot deal with. Given this, I think it is necessary to also consider design issues before coming up with requirements. We presumably don't want to specify requirements for which we don't have a good idea of how we could achieve them. (But maybe we do if we like adventure). So much of what follows will address high-level design ideas as a way of demonstrating that what we would like to do is in fact practical.

First, some discussion of what the potential new complexities are. What does it mean to provide the most appropriate calibration reference files? Appropriate for what? We shouldn't lose sight of the fact that the most recent calibration reference files may not be the most appropriate versions of the files for older versions of the calibration pipelines. A concrete (and plausible) example is appropriate.

If one comes up with a better calibration algorithm, it may result in the need for entirely new reference files to employ that algorithm better, even if the reference files are essentially of the same form and content they were previously (arguably, anytime we change the the form and content of a reference file, we should change its type).

If users are running older versions of the calibration pipelines on their home systems, should they be able to use those pipelines with CRDS? I think so.

3.1 Supporting multiple versions

The current CDBS presumes that the current set of reference files is appropriate for all contexts. We have already run into a case where that wasn't true. The context for the ETCs was not the same as for the pipelines, and that led to problems with updating the CDBS files for the ETCs.

In other words, I'm arguing the current CDBS model is inadequate even for our current operations, and more so for the way we would like it to work for JWST calibration pipeline software.

It should be possible to come up with different recommended reference files for different contexts. Those contexts may be different versions of software, or they may be different systems that require some divergence in which sets of reference files they wish to use. An example may be to try to replicate results that were generated at some point in the past without depending on the specific reference filenames being present in the raw data.

The reason it is a problem with the current CDBS is that a single database instance is used to determine the most appropriate files to use for a given dataset. While it is possible one might figure out a database schema that could manage different contexts, I'm going to argue that a database is the wrong approach to solve this issue. So if not a database, then what? I'll get to that.

3.2 Separating Reference Data and Mapping Info

There are other problems with the current approach used by CDBS that should also be discussed. The current system has a tight coupling between reference files and the modes and dates of applicability. A delivery of a reference file ties it to the modes and dates it should cover, and that coupling makes it difficult to change the mapping of referenc files to data without doing new deliveries of reference files. Particularly problematic is how dates of applicability are tied very tightly to the reference file though the use of the USEAFTER keyword. In that case, the date segmentation depends on the sequence of file deliveries and the value of the USEAFTER keyword. A single mistake in a delivery can require many redelivered files to fix the problem. I'm going to argue that we should be able to change how reference files are applied (e.g., for which dates and modes) without require new deliveries of reference files. Deliveries of reference files, and the specification of how they are to be applied should be completely separated. Doing this will be necessary to achieve the context dependency features I think are necessary. At the moment, the coupling between reference files and modes in the database prevents this.

The current representation also can make it difficult to understand what the current state is. The database contains the whole history of reference files, mode couplings, and USEAFTER dates. Determining which files are currently active and whether all modes are covered can involve a complex analysis of the database. This is made more difficult if many deliveries have been made. (The current system does allow summaries to be generated for which reference files are active and what modes they cover, but the point is that the database itself isn't necessarily transparent).

4 Considering a Different Approach

I think there is a different way of approaching the whole issue. What I'm not sure about is whether I'm missing some important capability of the current system that the new approach doesn't handle well. I could well be oblivious to some of the current needs or requirements.

If one can decouple reference files from the specification of which ones should be used then we can construct a system where there are multiple mappings possible, and these mappings can be made context dependent. Ultimately, what we are seeking is a means of dividing the parameter space (which consists of a subset of instrument configuration parameters, the date, and possibly data-dependent parameters) into regions that are assigned to the appropriate reference file. Ideally, such mappings should be more flexible than CDBS currently provides, and it should be possible to define more than one mapping.

Being able to support multiple mappings would help remove one of the common problems that have occurred in the past with CDBS deliveries. There have been numerous cases where new reference files have required a new version of the calibration pipeline, and the new version of the pipeline has required new reference files. This results in a case where the deliveries of the reference files and new software must be synchronized, and there have been a number of times where the synchronization has failed resulting in failures in the pipeline. Arguably some of these changes should have resulted in a new type of reference file, yet adding new types of reference files involves a fair amount of work in the current system, and understandably there is a reluctance to do so even if that really is the right thing to do (e.g., if the semantics of the reference file contents change, even if its form doesn't, is it really the same type of reference file?).

If multiple contexts are supported, then one thing that becomes possible is the delivery of reference files followed by a definition of a new context, before the software is delivered. Then when the new software arrives, the proper context is already in place, and for the previous version of the software running, it continues to use the older context. This would simplify such coordinated deliveries quite a bit.

4.1 Flexibility and conciseness for mappings needed

I don't believe that a database table is necessarily the best way to define these mappings. That is because it is possible that the mappings could involve mixtures of parameters regarding the boundaries defined. For example, one could base flat field assignments more simply by using effective wavelengths, where effective wavelength is derived from instrument configuration parameters. Currently this is typically done by assigning every possible combination of filters to a specific reference file, when one suffices for many. The database method precludes the use of floating point parameters for selecting reference files, at least in the current implementation.

In fact, there may be many different ways to define mappings, and using a table isn't always the most efficient or flexible way to define such mappings. We should allow a system to be flexible enough to define new mappings without making major changes to its architecture.

In particular, I envision a system where each reference file type has a corresponding file that defines each mapping. The form that the mapping definition may vary from reference file type to reference file type. New kinds of mapping definitions may be added as new reference file types are added. When one or more new reference files are delivered, then a new mapping file can be created that uses these new reference files. Different contexts can use different mapping definitions. All the mapping definition files are archived, and catalogs are created that records metadata about the reference files, and the mapping definition files. It is the mapping definition file that is used to determine the most appropriate reference file of that type that should be used for a specific situation. Each mapping scheme may have its

own software that uses the data to determine which reference file a particular data set should use. That is, if we find a new way of splitting up parameter space that is most efficiently represented in a different scheme, then we can add software that interprets that scheme. To use a simple and not necessarily realistic example, suppose we wish flat fields to be chosen by assigning an effective wavelength to each flat field (as mentioned previously). Then we may map all instrument modes to an effective wavelength, and then specify something simple such as:

```
GeometricallyClosest(("effective_lambda",),
   (1200., "ACSflatfield1.fits"),
   (2000., "ACSflatfield2.fits"),
   ...
   (8000., "ACSflatfield9.fits")
)
```

Corresponding to this mapping definition would be software that would retrieve the effective wavelength for a given instrument configuration (perhaps precomputed, or not), and determine which item in the above list was closest in the geometric sense and select that one as the appropriate reference file.

Another example might involve returning more than one reference file because of interpolation. Say for sensitivity:

```
LinearInterpolation(("date","effective_lambda"),
  (2017.2, "NIRCAMphotflam1.fits")
  (2018.5, "NIRCAMphotflam2.fits")
  ...
  (2021.7, "NIRCAMphotflam8.fits")
)
```

and if the dataset had a date of 2018.0, then the software would return the name of two reference files (the first two), with corresponding weights to be used with each in order to apply the appropriate interpolation for the values obtained from them.

Currently, the first case requires an enumeration of all modes with reference files specified for each one. This can result in a very long list, and make it much harder to check for accuracy if the original intent was as simple as shown here. The second requires special construction of reference files that have date built into the data (e.g., tables with date as one of columns), where a new version of the file is generated with the added date data that replaces the previous forms. This adds to the complexity of the reference file, and the greater chance that it will be corrupted with an update. With the form shown here, all one has to do is add a new sensitivity table for the latest date along with a new mapping file that adds that entry, and leave all the previous reference files untouched.

If the mapping definition files can generally be made simple (and I'm thinking that will usually be the case), then there is generally no good reason to use a database for the mapping definition itself. One saves each version of the mapping file, and several can be in use at any given time depending what system it is running on or what software is using it. Comparisons between version of mappings can use simple text comparison tools to see what has changed over time. One could even use a version control system as a means of managing the changes (while archiving all the files at the same time). This would reduce the special purpose software needed for the system. [But there are many issues to consider regarding such uses. We should ensure that all information is saved independent of the details of a version control system. We have already seen more than 2 generations of version control systems and it would be a mistake to hitch our system to the particulars of one. On the other hand, we can probably presume that future version control systems will be able to migrate history from older, popular ones, and have a superset of features of the older ones.]

It is also important to say that although the mappings themselves are no longer represented by database tables, the mappings could be saved in a database to be retrieved by the software that uses them. The database that catalogs all the mappings could store the actual text, or references to the file that defines the mapping.

I've glossed over one important aspect, since I think there are many possibilities that can be considered, and I don't claim to have the definitive answer on it. Namely [ahem] what naming scheme could be used for reference files and mapping files. It seems to me that the current scheme of generating obscure reference file names based on date/time

generators, is needlessly obscure. I don't see why simpler naming schemes (and perhaps more descriptive) couldn't be used. But this is probably not the place to delve into suggestions or alternatives. It's more important to consider whether the approach I am discussing is appropriate or not.

I'll also note that what I'm describing has greater similarity to how the synphot reference file system works than the calibration file system in that the components file is what defines what reference files are to be used. It defines the mapping. That scheme is complex because of the large numbers of reference files involved, and a simpler approach could be used by decomposing the existing files (using many different mapping files instead of just one, segregating by at least instrument, and perhaps submodes of instruments). In fact, I'd argue that the nonconformance of the synphot files (now pysynphot) is an illustration of the current inflexibility of CDBS to handle different mapping schemes.

One final comment is to address the possible criticism that the current system has the advantage of self-documenting reference files. Since they contain information about what their intended use is, they are more useful as a result. Perhaps. I think is is good to separate such metadata about the data in the file itself, from the file since that may change over time. I think a new system should allow editing or appending such information about reference files ("we now think we can use this file for xxx cases (which wasn't true at the time of delivery") after reference files are delivered. And retrieval of reference files could insert such information into the header (commentary type additions only!) as part of the archive retrieval process.

5 Managing many contexts

In the previous discussion I laid out a possible approach that I believed could support many contexts simultaneously. But the more serious issue is whether it is possible to easily manage many contexts. Examples will illustrate much better than dry abstractions. To start I'll give an example that would likely be unmanageable.

Before going into a specific example, I'll should give an example of how the contexts might be defined (there are many possible ways to do this, and I do not mean to suggest this is the only way or the best). The criteria I consider important in outlining a possible scheme is that:

1. reference file mapping definitions can be specified in a simple ascii file
2. such mapping definitions have a clear meaning separate from any software implementation, nor are they locked to any particular language
3. different instances of a reference file mapping is reflected in a unique filenames. That is, different versions of a reference file mapping will have different filenames.
4. reference file mapping definitions indicate as part of the filename, which instrument, and what kind of reference file is being mapped.
5. it is possible to define higher level contexts (e.g., sets of mappings for different kinds of reference files, by listing the name of reference file mapping definitions for each kind of reference file, and instrument. These contexts can be nested, e.g. We can have one per instrument, where the file for a given instrument lists all the relevant reference file mapping definitions for that instrument. And such instrument-level context files could then be referenced by a higher level context file containing a list of the instrument context files.
6. such higher level context files would have unique names. Instrument-level context files would have names that identified the instrument.

If we consider calibration pipeline versions to be the primary context (there could be others), and to take a specific instrument (let's call it XXX, and use HST terminology for familiarity's sake; e.g., CALXXX), we could be in a position of supporting 10 different version of CALXXX (v1, v2, ... v10).

So to use a concrete example, flat field reference files, the contents of a flat field mapping file could generically look like this (I'm leaving out lots of other info to convey the basic idea):

```
filename: XXX_flatfield_v003.rmap
ClosestGeometricRatio(
  'effective_wavelength',
  (1.2, 'cref_XXX_flatfield_123.fits'),
  (1.5, 'cref_XXX_flatfield_124.fits'),
  ...
  (5.0, 'cref_XXX_flatfield_137.fits')
)
```

And an instrument context that uses it

```
filename: XXX_context_v015.rmap
XXX_flatfield_v003.rmap
XXX_dark_v009.rmap
...
XXX_mdriz_v893.rmap
```

And the top level pipeline context file that contains that:

```
filename: jwstpipeline_context_v087.rmap
XXX_context_v015.rmap
YYY_context_v002.rmap
...
ZZZ_context_v042.rmap
```

So far, fairly straightforward. Any change in a lower level file, must result in new versions all the way up the chain.

But left out is how we manage contexts relative to pipeline software versions. For individual calibration pipelines, we can presume that there is an existing context file that is most appropriate for that software version. So we could have a table like this

```
CALXXX v1: XXX_context_v007.rmap
CALXXX v2: XXX_context_v009.rmap
...
CALXXX v10: XXX_context_v022.rmap
```

But is this really what we want to deal with? So with every reference file delivery for XXX we are forced to review whether we need to update any of the context for 10 different software versions. Will instrument scientists even be able to figure out easily which ones should be updated and which ones not. With every new update to CALXXX we increase the burden of what must be reviewed with reference file updates. Allowing this level of generality seems impractical. Is there a way of making it more reasonable? I think so. I will just suggest one way as an illustration of how such generality can be reasonably managed (an existence proof). There are probably many alternatives, and a wide spectrum of effort required in maintaining such schemes.

The usual pattern is that new calibration reference files apply just as well to all versions of the calibration software. This isn't always true, but it is usually true. We should take advantage of this to simplify management of reference file mappings. If we can designate for reference files, which versions of the software they can be used for, then perhaps we can come up a more generic mapping description that can be used for multiple versions of the software and preferably all versions of the software.

But there are occasions that mean that a reference file should not be used with certain versions of the software. Personally, I favor trying to keep the sense of a reference file type as consistent as possible so that backward compatibility is enabled. If the semantics of the file change, i.e., contents must be used in a different way, I'd argue that a new reference file type be created. New types indicate that the files really are not equivalent. It also simplifies what the calibration software must do with the files. If we add a new type, the old software won't care, since it won't look for it. And the new software won't generally care for the old type either. In the current CDBS system people have avoided adding new types since it involved much work [am I wrong about that?]. If the new system makes it easier to add new

types, then that should be the preferred way of dealing with changes in the reference file semantics. (I will address what could be done to enable easier additions in the future). A trivial (and probably silly) example of this is deciding to multiply rather than divide by the flat field.

Nevertheless, either by mistake, or for unavoidable reasons, there may be a need to support different reference files for different version of the software so we should consider a mechanism to support that. One way would be to provide a default, and possible alternatives that have version restrictions defined. Here is one (very quickly thought out) possibility using the silly example of inverting the meaning of a flat field after a certain version:

```
filename: XXX_flatfield_v004.rmap
ClosestGeometricRatio(
    'effective_wavelength',
    (1.2, SWVersionDep({'default':'cref_XXX_flatfield_123.fits',
                        '<5':'cref_XXX_flatfield_73.fits'})),
    (1.5, SWVersionDep({'default':'cref_XXX_flatfield_124.fits',
                        '<5':'cref_XXX_flatfield_74.fits'})),
    ...
    (5.0, SWVersionDep({'default':'cref_XXX_flatfield_137.fits',
                        '<5':'cref_XXX_flatfield_87.fits'}))
)
```

What this says is to use the first flat specified in each entry unless one of the other version conditions is satisfied (here only one alternative is listed), where if the CALXXX version is < 5, then it should use the flat specified for that condition. (There are many ways to do this and this is probably not the best).

By doing this, we can use the same mapping file for all software versions thus reducing the burden of dealing with many possible context files. In effect, the context for software versions is contained with the mapping file rather than the context files. But if for some strange reason, we had to specify different context files for other reasons, we could do so.

Time variation can be folded into such a structure as well. Suppose in addition to the changed interpretation of flat fields, there is a time dependence in flat fields that we must handle. Again, I'm sure there are better ways of doing this, but here is one alternative:

```
filename: XXX_flatfield_v005.rmap
ClosestGeometricRatio(
'effective_wavelength',
(1.2, SWVersionDep(
SelectClosestTime(
    '2017-4-24':{'default':'cref_XXX_flatfield_123.fits',
                 '<5':'cref_XXX_flatfield_73.fits'}},
    '2018-2-1' :{'default':'cref_XXX_flatfield_222.fits',
                 '<5':'cref_XXX_flatfield_223.fits'}},
    '2019-4-15':{'default':'cref_XXX_flatfield_517.fits',
                 '<5':'cref_XXX_flatfield_518.fits'}},
)),
(1.5, SWVersionDep(
SelectClosestTime(
    '2017-4-24':{'default':'cref_XXX_flatfield_124.fits',
                 '<5':'cref_XXX_flatfield_74.fits'}},
    '2019-1-1' :{'default':'cref_XXX_flatfield_489.fits',
                 '<5':'cref_XXX_flatfield_490.fits'}},
)),
...
(5.0, SWVersionDep({'default':'cref_XXX_flatfield_137.fits',
                    '<5':'cref_XXX_flatfield_87.fits'}))
)
```

This is starting to get messy (And yet, consider the alternatives. it isn't even possible to do this now in the first place, and if it were added, how complicated would it be to specify it?). Notice this is fairly flexible. Each wavelength can have independent time points, and some may have no time dependency at all.

There are still many other design issues to consider. Such as

1. whether one can use a version control system to manage these sorts of files.
2. validation issues for reference, mapping, and context files
3. making it easy to add new types of reference files
4. ensuring transactional security when making deliveries of reference, mapping and context files

6 Can a Version Control System be Used?

So why even consider a VCS? At this point, I have doubts about whether using one is practical. But I can see some definite advantages. I'll start with those, and then the many negatives or issues that would need addressing if we used one. I am not suggesting the VCS be used for the data files, but only for mapping and context files.

The advantages:

I think there is one large advantage in that it provides a lot of built in machinery for tracking differences between versions. And to extend that, tools to do branching, tagging and merging may be very useful as well. It is possible that unique naming issues could be handled automatically by a VCS too.

The disadvantages:

1. We must not depend on any particular VCS. We've already seen two generations of VCS come and go (RCS, CVS). SVN may be replaced by distributed VCSs. So all information must be mirrored elsewhere (e.g., the archive).
2. Naming issues. The advantages of the VCS are when we use it to track versions of the same *named* file. Yet we want these files to have different names outside of the system. So there would have to be some sort of mapping scheme. The file type is the VCS name, but hooks are built in so that when files are exported, a name with a version number is attached. But how versioning is done within a VCS varies from one system to another (e.g., CVS and SVN differ greatly). I'm not sure people would appreciate the SVN numbering system for versioning mapping and context files. And this numbering scheme may not work well with others (git for example). I think this is a fairly serious problem. Perhaps one solution is to use hooks to insert version numbers into each file in some sort of keyword, that the export hook will use to build the filename.
3. Instrument scientists probably don't flock to version control system tools or find them particularly intuitive.

And so I'm already discouraged enough that I think I would put this idea on the back burner.

7 Validation Issues

An important part of such a system is ensuring that the files delivered are considered valid. I'll start backwards and list some of the most important validations needed. Going backwards means starting with the simplest tests.

7.1 Unique naming:

There is one generic issue and that is ensuring that the file has a unique name. I don't think (or at least I hope) we need to use the random naming scheme currently used. My understanding is that naming scheme was based on generating a unique name driven by the time the name request was made some some time resolution, i.e., so long as two requests for a name were not made close enough together in time. Presumably other means of ruling out name collisions weren't

considered practical at the time. I sure hope that isn't true now. I'm going to assume that the archiving system will provide a safe means of checking the uniqueness of names. I suppose the issue is making sure that one can check for the availability of a name, and then locking it for use. This seems like a very common problem that I'm sure has been solved many times in database systems and applications.

Should the names be auto-generated? For example, automatically incremented version numbers. Or manually supplied, and just checked for uniqueness? Or some hybrid (you can pick a number, but it should be larger than the last submitted one? For simplicity, I probably would lean towards automatically generated, but I don't think it is a critical design issue.

7.2 Context files:

1. Context files should be checked for correct syntax (through evaluation of the context file)
2. That they included all required metadata (haven't specified that yet)
3. That all mapping or context files they refer are in the archive catalog for such files.

7.3 Mapping files:

1. Mapping files should be checked for correct syntax (through evaluation of the mapping file, e.g., it should be executed and shown to properly generate mappings.
2. That they included all required metadata (haven't specified that yet)
3. That all reference files they refer are in the archive catalog for such files.
4. That the reference files are the right type of files for the mapping type. There are two levels possible for such checks, one is a superficial one that checks that it is the right kind of reference file; a deeper check would be to see that the selected reference file is compatible with the mode. At some level one might be required to run the appropriate calibration step to verify that, or that the calibration code use the same validator routine that the reference file system uses. We probably don't want to make the deep check a requirement, but it would be a nice goal to achieve.
5. That the mapping pass existing regression tests (more discussion on what these tests should include later)

7.4 Reference files:

1. Reference files have the appropriate file structure for that reference file type. I don't think reference files should necessarily be restricted to FITS (FITS should be used for array and tabular data). We have examples for the ETC that are better specified in some sort of ascii format.

For FITS reference files:

1. Reference files have the expected extensions (if any)
2. All headers have the required keywords (required for that reference file type, not by FITS)
3. Primary and extension image data components are of the right dimensionality and data type, and match any specific value restrictions (e.g., only positive values) for that data item.
4. FITS tables have the required columns with the required types and dimensionality (and with possible value restrictions)
5. FITS tables may have optional columns, but such columns must be specifically permitted by an explicit list of optional columns (this allows for the augmentation of data as long as it doesn't change the meaning of required columns)

For other kinds of reference files:

1. TBD system of verifying valid contents

7.5 Discussion:

Regression tests for mapping files:

There are various issues to consider in doing tests on such files. We could pre-define tests to see if it handles a specified set of modes/dates/etc, that is, gives a result. That raises the issue of what the right behavior is when no match is found. Should it raise an exception (I think so), or should we always provide some sort of default if no other match is found (generally not, but in some cases yes). We could use the history of all previous exposures as a test set, but this seems like it would inefficient to run regularly. But I can see utilities to cull out from the exposure catalog all unique cases for subsets of mode parameters for generating regression tests. This could be done every so often to update a test suite. The main purpose of testing here is to see that matches are made, not what the match is (since that is supposed to change when you submit a new mapping file!). It could also test that the regression file works with the parameters that selected it (again, probably should not make this a requirement, but a good goal to have).

Reference file validation:

This is not particularly simple given the many varieties that may occur. We should have a fairly efficient system for specifying the expectations for reference files so that the tests are driven directly from these specifications, and that the specifications also serve as the documentation (I am not in favor of maintaining a document, and parallel code; the specification should be used to generate both the documentation and the tests). If that means the specification looks like a program, then it looks like a program. It needs to be efficient as to make adding new reference file types as simple and efficient as possible (the barrier for doing that must be reasonably low).

When mapping and reference files are committed to the system, the version of tests for each will be recorded. The tests and specification files will also be archived as part of the system.

Adding new reference file types:

Adding new reference file types should be as data driven as possible. Generally it should not require new code be added to the reference file system to add a reference file type. I would envision the process of adding a new reference file type to be:

1. Some sort of TBD approval process that permits the creation of a new reference file type, and a name for that type.
2. A new type is registered with the system, possibly by adding a new row to a database table of reference file types, with appropriate associated information that includes:

1. Motivation, rationale, and description for that type, and testing history, ISR refs, etc
2. The specification file for that type. (used to generate documentation and validation tests)

1. Any updates to the specification file, will be automatically tested against all existing archived instances of that reference file type to ensure that they are all consistent with it. This allows adding information (e.g., new optional columns), or loosening restrictions (data that was integer only before may be integer or float), but not changing previous specifications.

Note that this does not include mapping criteria!

8 Transactional integrity issues

I believe we can largely rely on the archive and reference file catalog for this. All relevant files used by the system should be archived. This includes:

- reference files
- mapping files
- context files
- reference specification files

This will require a multistep commit process (or a system that ensures the commits are sequenced in the dependency order, and only moves on when all previous steps are complete). Mapping files can't be committed until the system can verify all the reference files that are referred to are already part of the system. Context files can't be committed until it can be verified that all the context and mapping files they are referring to are already part of the system. Such checks may make such commits somewhat slower than they could be, but this should not be a big problem. The catalog(s) that results should never grow to a large size as catalogs go in this age.

The catalog(s) will also be used to ensure uniqueness of names.

One possibility is that the catalog is also used to lock names for files under development. The point that was made for this was it would be useful for the pre-commit testing of the file to have the same name as it will have in the system. This necessarily means a two stage operation of requesting and reserving a filename and the machinery to go with that.

All operational systems and web services should always obtain their files or the names of "best reference" files from the calibration reference file system and never rely on manual changes to their local copies of files. And the system itself should have safeguards in place to ensure the files that it delivers are consistent with the catalog and archive (it should make it difficult for someone to manually corrupt any file cache if such exists).

9 Frequency issues

What I mean is what about cases where a great many reference files maybe generated. A good example would be daily darks. What are the consequences of such frequent updates and how would these be handled?

9.1 Sizing issues:

What's the upper limit on how many reference files of a given type we may require? Daily darks are probably the example to date of the largest number we might encounter. If it were one file per day, and the mission lasted 10 years, then we would expect over 3600 files. Could we have more? Perhaps if we needed daily flats at many wavelengths. That seems unlikely, partly because the time to get daily calibrations is likely to be limited. So something on the order of one file per day seems like a likely limit.

What are the consequences of that many files? As far as bulk size of mapping files goes we would effectively see a mapping file grow by one line per day. Even if the line is very long, the final file would not likely be more than a few megabytes (1K bytes/per line!). That is quite manageable by today's standards. But we will have 3600 of these files so the net size of all these files would be in the GB. Annoying, but actually not much space these days. Even so, that could be reduced by allowing mapping files to use an include mechanism of some sort. For example, every month or year, we create a new include that lists all the daily darks for that period. Once the month or year is full, that file never changes. The active mapping file has includes for all full periods, plus a list of the daily files for the latest, incomplete period. But it isn't clear that such a mechanism is needed. It would make visual inspection of the mapping file much easier though.

9.2 Speed issues:

Does doing bestref matching on such a mapping list indicate significant performance problems (as opposed to using a database)? I'd be very surprised if it did. Such a data structure should fit very easily within memory, and be retrieved very quickly using current look up mechanisms. This should not be a significant issue.

9.3 Update frequency issues:

Perhaps of the greatest concern is that in what I've described to date, every daily update to a reference file means there are required new versions of context files up the chain. That could be a burden depending on how these files must be generated and managed. Is this done automatically in most cases? Manually seems out of the question, unless the updates are done less frequently then daily (e.g., weekly or less frequently). If they are done automatically, then by what rules? We generally do not want to pick up all new reference files. Automatic updates would presumably be only on a very specific subset of reference files that we know are updated automatically. Even now, I think such frequent updates require manual release (is that right?).

Another possibility is to allow context files to use a mechanism to specify that they wish to use the most recent mapping file available. With that, the context file does not need to be updated frequently. There are two drawbacks though. The first is it complicates a bit knowing what reference files one did use on a specific date. The previous scheme is dead simple in that regard. Everything is explicit. With references to the latest available, then one has to query the database to see what the latest available was at that time. With the current scheme, that query should be very simple. The complication is then one needs to understand how the operations system does its updates. In the original scheme, the update to its context is explicitly done. When references are made to the latest available, then either the operations system must periodically re-initialize (hourly, daily, etc). Now knowing what was used requires knowing information in the catalog, and the policies of the operations system that was running (e.g., when it does its updates)

Another complication is with regard to testing. We wish to be able to commit new references files, and mapping files without their going into production so that they can be tested in as close to a production environment (something that is difficult now). The mechanism that says "use the latest version" defeats that kind of approach. To allow testing, we would have to have the catalog give the files some state with regard to being production ready or not (something that has been talked about for the current CDBS). Then the model is you deliver files, do testing, then change the state of the file to production-ready after one is satisfied.

All in all, how frequent updates are done needs more thought and analysis, I don't believe it is necessary to decide one way or another. I think either approach is probably workable.

9.4 Experience with sizes:

I did do a quick analysis of HST reference file state. I basically counted up the total number of lines for each type of reference files. Each line generally corresponds to a supported mode. In some cases many lines map to the same reference file so one should not interpret this as the number of currently active reference files and especially not the total number of reference files submitted. The results are shown in Appendix A

10 Handling pysynphot Reference Files

Where do pysynphot files live in this proposed scheme?

I've always been a bit troubled by the disparate ways that pipeline calibration reference files and synphot (now pysynphot) reference files have been handled. The current CDBS treats the two in very different ways both in the code that handles them, and in the procedures required to manage them.

Anything that makes these files handled in more uniform ways would be a great benefit in my opinion. Nevertheless, there are significant differences in how these files are used by their respective software that is the source of many of these differences. Can these different use approaches be addressed by the scheme I'm suggesting?

First is it worth reviewing why the differences exist. For calibration software, typically there is a need to select between different reference files based on a set of parameters. That could include instrument mode and the date. But typically each reference file type is treated individually. Flat fields and darks don't need any sort of association together. The code that corrects for flat fields generally doesn't care about darks, and the code that corrects for dark counts or current, doesn't care about flat fields.

But synphot uses a large ensemble of reference files together. To date, it never picks between different files based on selection criteria. Actually, that's not quite true. In a way it does, but it does that based on the graph table it is using; for example, if you want an ACS computation, it know which transmission files to use based on the configuration you give it. In other words, synphot internalizes the mapping function with regard to instrument configuration (i.e., selection criteria). It only uses CDBS to give it the latest version of each throughput and DQE file and really little else. It doesn't even use CDBS really. It picks the appropriate top level files (graph and component tables), and they point to everything else. CDBS is just used to keep a history of file deliveries for the most part and ensure the files are archived.

How should it work in the new scheme I'm suggesting? There are a couple alternatives. In many respects, the graph and components file are the equivalent of a mapping system. We could generate an equivalent set of context and mapping files that do the same thing. In that view, the generation of the sequence of needed files from the context and mapping files would move outside of pysynphot, and pysynphot would be provided a list of files to perform the calculation on after it provided the bestref machinery the configuration information (telescope, instrument, mode, date, etc). That approach would be most in the spirit of how the calibration pipelines would use the reference file machinery.

A different approach would be to use the mapping files to map the specific throughput element (e.g., a specific ACS filter throughput) to a specific file (there may be multiple versions of that file). Currently that function is done by the components table. One could have one mapping file per throughput or detector. That would result in many files and seems a bit silly. Is there any reason mapping files could not handle sets of files, particularly if they have a good reason to be associated? That's something that should be considered. If all the throughput files for an instrument are mapped in one file, then that would closely correspond to the way we want existing component files to be broken up.

Something that has been glossed over so far is how files are tied to mapping files. If they were always one to one, then the name of a mapping file could be associated to the reference file being mapped. But if we want to use the same mapping file for many reference file items, that won't work unless we use a naming scheme for the file that is matched to part of the reference file naming system. For example, if all pysynphot reference files were named something like this:

```
cref_syn_hst_wfc3ir_f110w_v007.fits
```

then the mapping file for wfc3 might be named:

```
syn_hst_wfc3.rmap
```

And that that name means that it handles all reference files that begin with that name (with cref). But I'm not crazy about relying too much on filename conventions.

In any event, one could have a mapping file declare a manifest of what reference files it handles and has a separate mapping section for each file type. There are issues about how context and mapping files are handled by the bestref machinery (I'm presuming that the files are followed down the whole hierarchy to the endpoint mapping files, and thus all the reference files that are handled are discovered).

Note that this approach does allow for handling time variability through the mapping file rather than putting it into the reference file.

Used in this way, pysynphot files are handled using essentially the same machinery that the calibration pipeline uses. In many respects, the new scheme is closer to what pysynphot does now.

11 Language Dependencies

No doubt many would have noticed my examples of mapping schemes look a lot like Python syntax. I can hear you say: "Perry, you lying weasel, you've trying to trick us into depending on Python".

Not true. Really. I use it in examples since I know that it can work that way. But I think an important principle of mapping definitions is that they be easily interpreted by any software. They should have simple syntax and be easily parsed. Indeed, even if the software were written in Python, I would force the tool that read the mapping files to parse them rather than directly execute them to ensure that someone wasn't sneaking into the definition Python constructs that were not part of the mapping definition syntax. The syntax should support strings, integers and floats, and express relationships (like how to interpret the data, e.g., use nearest item, interpolate, etc.), but not much more. At some time, we may see that there is a use to allowing more general programming constructs. But there better be a very compelling reason, and it should be resisted.

And anyway, it's probably better written in Perl or INTERCAL.

12 Some Useful Tools

Diffs between two specified versions (and if versions weren't provided, the last two versions) of:

- reference files (header keywords and summary of data diffs via statistics)
- mapping files
- context files

Formatting:

To make diff'ing more useful, it probably is a good idea for committing context and mapping files to apply a formatting ('pretty-printing), to standardize indents, line-breaks, strip meaningless white space, etc. in a way that will tend to make comparisons between versions not show meaningless format changes. (or perhaps the diff'ing tool could apply such formatting consistency?)

Tools to examine changes at a higher level

After updating mapping and reference files, it would be good to have a tool that shows what the current reference files being used are (to make sure the ones you just delivered are in fact being used). This is more than just best ref, it is more a way of displaying all or some of the mapping specifications that result from a given context file; i.e., does that context map into the new reference files.

It should also be possible to compare the difference in the mapping specifications between two context files for all or some of the reference file types to see what changed. In effect, a diff that will follow the chain of files and compare the endpoints.

Tools to update context file chains?

Suppose I update a set of reference files and the associated mapping file. I would like to update the context files up the line as not to do that by hand. Some fairly easy way of doing that should be possible.

Tool to list all reference files used by a context

One should be able to give a context file and optionally a type of reference file, and and list all reference files used by that context.

Tool to track what contexts specified reference files are used by

One should be able to list all context files that use a specified reference file. This is not simple to determine in the proposed scheme at it would require searching all contexts to see if they refer to the file. But there is a simple solution that does make this an efficient search. A database can be created that adds a record for every context/reference file pair. Once a context is created, this database is updated with the appropriate rows. Furthermore, this row never needs

to be modified. Efficient searches may be performed on this database to see which contexts are connected to a specified reference file. [tool requested by Rosa Diaz]

Tool to determine which observations in the exposure catalog would be affected by a context change [requested by Rick White]

This kind of tool would help limit the scope of batch reprocessings as currently envisioned for DMS. This essentially requires comparing the reference file (for each type of reference file) used by the previous processing with the reference file that would be used by a new context. The question is how long would it take to perform such an analysis under the proposed scheme. In other words, how long does it take to perform the bestref functionality on upwards of a million exposures (close to what HST has). Does a database representation for the mapping make this more efficient rather than the proposed scheme? It isn't clear how efficient this is even with a database representation as I don't think anyone does this now (am I wrong?).

In any case, I don't think this is a problem for the suggested scheme. No doubt that some mapping algorithms could be too slow to be used in this way. But many aren't. I've been able to read all WFPC2 exposure records from the HST catalog within 2-3 minutes over the network. The size of the catalog is such that it is easily held within memory. All the mapping rules we have now for HST reference files can be executed on all the WFPC2 exposures within a minute or two once the information is in memory. The proposed scheme will be sufficiently fast for anything the current scheme supports. Besides, there are many mapping algorithms that will be sufficiently fast that are not easily supported by the current scheme. One needs to process 100,000 exposures within 10 minutes or so to be considered fairly responsive (though I can see it taking longer, particularly if one can split the job between CPUs). That means being able to evaluate maps at the rate of approximately 170/second, which isn't particularly fast. We can require that any mapping algorithm be sufficiently fast so that we can perform this function. Just because it is possible to devise some that would be too slow doesn't mean that we can't use this scheme in general.

Even that isn't as severe limitation as it seems in some cases. Take a semi-reasonable example and illustrate the complexities that might be present. Say we use effective wavelengths as the lookup mechanism for flat fields. Mapping by effective wavelength is going to be quite fast as far as software goes. But suppose we change the instrument's throughput curves. In principle, this is going to change the effective wavelengths for all observations. Worse, computing the effective wavelength is comparatively expensive (a pysynphot call) and could take a significant fraction of a second. There are two possible solutions. We could say: true, but the changes to effective wavelength are likely to be very minor unless the throughput changes are gross. We save the effective wavelength in the header, and we explicitly don't recompute these unless we ask to. So no reprocessing is done in this case.

But really, this isn't a problem if you do recompute effective wavelengths. Nothing keeps us from caching results in a catalog search. The reality is that there aren't that many different optical configurations in the catalog. One only needs to compute new effective wavelengths for a limited number of cases. These will take a little time, but once cached, go very quickly.

13 Draft Requirements

The purpose of all this was to convince myself that some of the requirements that we might consider for a calibration reference file system were actually achievable. Whether that requires a completely new system or an updated CDBS is really beside the point as far as the requirements go and don't need to be addressed here. I do believe the following requirements are practical. I'll say a few words about an oft requested feature that is not practical.

1. The CRDS shall identify the most appropriate reference files to use for the calibration pipeline.
2. The CRDS shall be able to do so for multiple versions of the calibration pipeline simultaneously. Namely it shall be able to identify different sets of appropriate files for different versions of the calibration pipeline. This will allow systems running different versions of the calibration pipelines (e.g., for testing purpose) to obtain the most appropriate files they need).
3. The CRDS shall be able to support committing new reference files without affecting operations so that new reference files may be tested in an operational context without being used immediately in operations.

4. The CRDS shall be used locally and remotely through a web service mechanism.
5. Outside users or systems shall be able to query the CRDS for the most appropriate reference files for the specific version of a calibration pipeline and a particular observation data set.
6. The CRDS shall be able to recreate the list of identified appropriate files for any point in the past given the date of the request, version of software, and a specific observation.
7. The CRDS shall record meta data about the reference files and all information regarding how observations are matched to reference files. Such meta data shall include (but not limited to):

1. How the reference file was generated (or links to documents regarding that)
2. Who created the reference file
3. Who committed the reference file
4. Information about the significance of the change

It shall be possible to augment such meta data after the fact along with information about when and who added the new information.

1. It shall be possible to mark committed files as bad, and that they should never be used, so that any attempt to identify them as an appropriate file will result in an error.
2. Rules about how files are matched to data and software versions shall be completely disassociated from the reference files themselves. In other words, it will be possible to change how software versions and data are matched to references files without requiring reference files to be resubmitted if no changes are needed to the references files themselves.
3. It shall be possible to show differences between the rules for how references files are matched to data between any two specific versions of such matching algorithms.
4. It shall be possible to easily undo the effects of a submission of a bad reference file or rules about matching without requiring new submissions.
5. Reference files shall include in their headers information regarding how they were created.
6. Reference files may include in their headers about their intended use, e.g., which data they are intended to be applied to. But such information is not guaranteed to be used by the actual rules used to match data to reference files.
7. It shall be possible to easily show the set of active reference files being used for all supported versions of the software, or a specific version of the software.
8. The CRDS shall ensure that all reference files and information regarding the rules for mapping data to reference files are safely archived before the files or rules are used by users of the CRDS.

Impractical Requirements

Any sort of automatic system for determining the effect of a reference file change on the science being done with the data. This is a *very* difficult problem and perhaps impossible to solve. While it may be a good idea to include metadata with reference data that generally categorizes the impact there is no foolproof system that can do so based on the data itself.

I'm certainly forgetting some things here, but at least it is a start

Appendix A .

These numbers were obtained by counting the number of lines for each reference file type from the INS web page that show what reference files are currently active (i.e., could be selected by bestref now). I did this a couple weeks ago so things have changed since then. Multiple lines may map to the same reference file (so when you see a comment about

many duplications, it means there are far fewer reference files than lines. Each line basically corresponds to one set of selection criteria, i.e., how many different cases are being tracked by the system. I don't think I was entirely consistent about the naming of the reference file, but there should be enough information to properly map them.

The link to the INS pages is:

http://www.stsci.edu/hst/observatory/cdbs/ and click on the instrument names at in the left hand side menu bar to access the different lists for all the instruments. These pages use direct calls to the CDBS database to get the listings.

```
WFCP2 ***********************************************************************************************

MASK: 6
ATOD: 4
BIAS: 53
DARK
  post2005: 738
  2005: 235
  2004: 234
  2003: 232
  2002: 252
  2001: 276
  2000: 273
  1999: 267
  1998: 294
  pre1998: 831

FLAT: 737
SHAD: 6
WF4T: 0
IDC: 3
OFF: 2
DGEO: 4
SBIAS: 6
SDARK: 12
SDELTADARK: 44
CFLAT: 41
PIXAREA: 1

STIS ************************************************************************************************

BIAS: 1141
DARK:
  CCD: 1000 (many duplications)
  MAMA: 16
PFLT: 1000 (many duplications)
DFLT: 0
LFLT: 719 (many duplications)
SHAD: 0
SDST: 0
ATOD: 0
APDS: 19
APER: 729 (very many duplications)
BADPIX: 5
CD: 58 (duplications)
CRREJ: 3
DISP: 5
GAC: 0
INANG: 4
IDC: 3
```

```
MLIN: 2
LAMP: 4
MOFF: 2
PC: 27
SDC: 9
CDS: 4
ECHSC: 2
EXS: 2
HALO: 2
RIP: 2
SRW: 2
TDC: 1
TDS: 25
WCP: 1
SPTRC: 10
XTRAC: 5

NICMOS ***************************************************************************************

BADPIX: 10
READNOISE: 203
DARK: 323
TDDARK: 9
LINCORR: 21
FLAT: 245
TDFLAT: 114
BACKILL: 57
PHOT: 6
GEO: 3
PERSISTMOD: 1
PERSISTMASK: 1
NONLINZERP: 2
NONLINPL: 2
SDSAACLEAN: 3
SAADARKS: 0
GRISM:?

ACS ******************************************************************************************

  BIAS
      (detector, gain, amp, useafter, xsize, ysize)
      WFC
          post-SM4 subarray comb: 1
          post-SM4 non-def amp: 12
          2006-2009 non-def amp: 16
          post-SM4 full frame, 4amp: 24
          2006-2009 full frame, 4amp: 35
          pre 2006: 310
      HRC
          post 2006 non-def amps nonstand gains: 13
          post 2006 def amp, std gains: 40
          pre 2006: 208
  CRREJ
      (detector, useafter)
      2
  CCD
      (detector, useafter)
      10
```

```
  IDCTAB
      7
  SBCLIN
      1
  DGEO
      33
  BADPIXEL
      6
  MULTIDRIZ
      3
  CCDOVERSCAN
      2
  ATOD
      104
  PFLT
      WFC
          polarizer: 96
          normal imaging: 30
          ramp filters: 5
      HRC
          coronograph: 47
          polarizer: 87
          normal imaging: 49
      SBC
          normal imaging: 20
 CORONOSPOT
      1
 CORONOSPOTFLT
      5
 DARK
     WFC: 1401
     HRC: 1343
     SBC: 2
 HRCEARTHFLT
     15

WFC3 ********************************************************************************************

BIAS: 64
DARK:
  UVIS: 15
  IR: 161
PFL
  UVIS
      BIN 1: 300
      BIN 2: 64
      BIN 3: 64
  IR: 34
DFL: 0
LFL: 0
SHD: 0
LIN: 3
BPX: 8
CCD: 4
OSC: 3
CRR: 3
MDZ: 2
PAM: 0
```

```
IDC: 2
DXY
A2D
FLS

COS ********************************************************************************************

FLAT: 3
GEODIST: 2
BADTIMEINT: 1
BASELINEREF: 1
BURSTPARAM: 1
1DEXTRACT: 8
BADPIX: 3
DEADTIME: 2
DISPCOEFF: 6
CALLAMP: 4
PULSEHEIGHT: 2
PHOTSENS: 4
TIMEDEPSENS: 2
WAVECALPARMS: 4
```