JSOC Dataset naming convention for use with the DRMS

version 7.0, 9 March 2009

Data is stored in the HMI/AIA JSOC in many "Data Series." A Data Series, or more commonly dataseries, is basically a sequence of like data objects, typically "images" or other binary data along with associated meta-data. Specifically, a dataseries consists of a sequence of Data Records, more commonly datarecords, or even more commonly just records. Usually, each datarecord is the data for one step in "time". Most but certainly not all dataseries are sequences in time. They can in principle be any list of data objects. It may be helpful to think of a dataseries as a table of rows and columns where each row represents a single record, and the columns correspond to elements of meta-data or binary data.

The actual storage methods for the meta-data and array data does not matter for the description of the naming conventions but some details will help in understanding the use of the naming system.

A datarecord consists of Keyword tagged meta-data describing the record and 0 or more named Data Segments (more commonly segments) usually containing binary arrays of data values. All datarecords in a given dataseries have the same set of keywords and segments, each having an associated name and type. The dataseries description and the datarecords are maintained in a relational database called DRMS (Data Record Management System). DRMS is implemented using a set of PostgreSQL tables (see http://www.postgresql.org).

Properly speaking, DRMS contains only a description of the data corresponding to each segment in every record. The binary data itself is stored in “Storage Units”, which are simply directories on disk or tar-files on tape. Storage units are owned by SUMS (Storage Unit Management System), which maintains tables in PostgreSQL to connect a storage unit number (or sunum) to its location. A storage unit will contain data for all segments for 1 or more records from the same series.

In summary:

A **Dataseries** consists of a set of:
Records each of which contains an instance of:
Keywords including sunum (if applicable), and
Segments each of which consists of structure information for data.

Records may also contain links to records in other series, but for simplicity this feature will not be explained in this document.

Normally one or more keywords are designated “primekeys”. The primekeys must together uniquely identify a record and are used to define the main index for the series. Any records with the same set of primekey values are assumed to be different versions of the same record and normally only the most recent version of any record is easily available. Thus the current version of any record in a given series may be found by specifying the values of the

primekeys for that series. All series have one pre-defined keyword called "recnum" (record number) which has a unique value for each record in the dataseries and is used for the main index in the case that no primekeys are defined. The implementation and implications of the version system is described near the end of this document.

The set of keywords and segments contained in a given dataseries, as well as which keywords are primekeys and which are indexed in the database, are usually set when the dataseries is first created. We will mention in passing several aspects of creating dataseries, but for a full explanation of JSOC series definitions, including a description of links between dataseries, please refer to http://jsoc.stanford.edu/jsocwiki/Jsd.

In order to access a set of records from a dataseries ones must query the database. We call that database query a "Dataset Name", or simply a dataset. The DRMS dataset name rules have been designed with the intention of providing user friendly names that are easy to remember and use. Hence database queries are actually descriptive names for datasets.

This document is concerned with the identification of datasets. Not so much with the logical or physical layout of the data or its storage or use but simply with naming conventions and the use of those naming conventions to identify specific data for purposes of processing.

At this point we must unfortunately introduce a slight ambiguity of terms. Since in DRMS a dataset is a set of datarecords selected by a query, we call the results of that query a Record Set, or more commonly a recordset. Hence the terms dataset and recordset will sometimes be used interchangeably. For the purpose of explanation, however, we shall restrict the definition of recordset to mean a set of records from a single dataseries. A dataset is then a collection of recordsets, possibly from different dataseries or even from different catalogs (see below). It should be born in mind, however, that as seen by the programmer, a dataset will be contained in a single DRMS_RecordSet_t structure comprising an array of records, regardless of their “recordset” affinity.

Most recordsets are expected to consist of records from DRMS, but the programmer’s library and by extension the naming rules do allow for access to data in a few other systems. To help differentiate the name rules for the different sources we call each source of recordsets a “Catalog”. Thus the primary catalog of the JSOC system is DRMS. The predecessor to DRMS, the Data Storage and Distribution System (DSDS) is another catalog. The unix/linux file system can also be viewed as a catalog and the JSOC recordset naming scheme allows for datarecords to be specified as simple files or directories of files in particular standard storage protocols (presently only simple FITS files are supported in this way.)

The naming rules provide several approaches to end up with a dataset. The “name” can be (and normally is) as simple as a single query resulting in a compact list of one or more records from a single series or it can be a list of such specifications or it can be a file name where the file contains a list of such specifications.

In the case of a simple query, dataset name should be in such a form that it can be published in documents in a way so that the same data (possibly of later version) may be extracted at a later time.

The JSOC DRMS naming scheme is described here as a formal grammar. The following syntax is described using a BNF[1] format where entities are in angle brackets "<>" and literals are not enclosed except isolated non-alphanumeric characters which may be enclosed in "". Variants are separated by "|". Optional elements are in curly brackets "{}". Parentheses "()" may be used to group variants to make precedence specific. Recursion is explicit where allowed. The common use of square brackets "[]" for single options is not used here to avoid confusion with the use of "[]" as literals. Each entity is defined after it is first used. An ellipsis "..." is used to show either an incomplete list or a set of any text depending on context. "::=" is pronounced "is defined as a". "C"-like notation is used for non-printing characters.

Importantly, these BNF grammatical clauses describe C strings. This implies an array of characters followed by a NULL terminator. The NULL terminator itself is not explicitly shown in any of the grammatical clauses that follow, but it is implied. In many cases, the C strings being described are strings that are typed into a terminal. However, in some cases, the C string may be the content of a file. In those cases, the end-of-file character is explicitly shown to impress upon the reader that the string resides in a file, and not on the command line.

A full DRMS name specification is at: http://jsoc.stanford.edu/jsocwiki/DrmsNames

Catalog

The term catalog is used here to denote an entire data collection. All of the MDI and solar group data managed by DSDS is part of the DSDS catalog. Similarly all of the HMI and AIA data managed in the JSOC at Stanford will be part of the DRMS catalog. It is possible that new catalogs will be added in the future, such as Virtual Observatory Tables, and as already mentioned, the file system constitutes another catalog. Each catalog will have its own syntax for specifying recordsets, described below.

For those familiar with DSDS, some of the similarities and differences between it and DRMS will be mentioned in the sections that follow. For those who know nothing about DSDS, these comments can be safely ignored. Be aware that terms like dataset do not mean the same thing in the two systems.

Series

DRMS dataseries names consist of two parts separated with a "." (period character aka "dot"). The leading part is a project name and is implemented as a Postgres namespace.

---

[1] There are many "BNF" formats. See e.g. http://www.cs.man.ac.uk/~pjj/bnf/ebnf.html. Therefore the conventions used here must be explicitly defined.

This means that access privileges can be set for all dataseries in each project independently. The space of names in this first section will contain some reserved names, such as mdi, hmi, aia, wso, etc. For these reserved names the default ownership and permissions will be maintained in a table of reserved project names. Individual users each have their own project names which by convention consist of two parts with the leading part indicating their group or institution and the latter part their specific identification, separated by an underscore. For example, one project name is "su_phil", where su means Stanford University, and phil is a personal identifier. The personal identifier is independent of both database (PostgreSQL) username and login userid, although it is common to find all three to be the same. The series specifier part of the name (after the dot) should indicate the observable, the cadence, the mapping or region size, the reduction level, etc., as appropriate. For example, a complete series name might be "hmi.fd_V_50s" for HMI full disk Dopplergrams at a 50 second cadence.

A DRMS dataseries has a "prime index" which is defined in its series definition by a list of keyword names or defaulted to the absolute record number. These are the primekeys (see above). The attributes named are used to construct a database index to allow rapid location of records identified by a value of its prime index. An example might well be using the attribute "T_OBS" as a prime index. In the case that the prime index keyword value is a real number (floating point type) it is often convenient to define a discrete mapping to a "slotted" axis to remove the uncertainty of exact matching of imprecise values. Slotted prime keywords are discussed later in this document. One may also specify keywords in addition to the primekeys to be indexed; please refer to http://jsoc.stanford.edu/jsocwiki/Jsd.

DSDS dataset names had the form "prog:XXX,level:XXX,series:XXX[NNN]", where the prog part played the role of project name, and the level and series parts together would give the information contained in the series specifier part of a DRMS dataseries name. The number enclosed in "[]" was an integer series number that was the sole index for the series. As in DRMS, multiple versions of data were allowed. Old versions could be retrieved by specifying a level number in addition to the series number. A record in a conforming dataset in DSDS could contain at most a single data array, whereas DRMS records are allowed to contain multiple segments, each of which corresponds to an n-dimensional array of data.

Dataset

A dataset is a list of recordsets. A dataset name is a description of that collection. In the description here the term "dataset" is used interchangeably with "dataset name" where the meaning is obvious from the context. A dataset may contain records from one or more dataseries. A subset of the dataset that can be simply described as a group of records from a single dataseries is called a recordset. With this description, to specify a dataset on the command line, in a file, or in code (in memory), one specifies a collection of recordsets separated by a semicolon, a comma, a comment in the form of "#" through to the first instance of another "#", a comment ending in a newline character, or a newline character. The following grammar makes this explicit.

<dataset> ::= <dataset> {<dataset_sep> {<dataset>}} |
<dataset_listfile> | <record_set>

<dataset_sep> ::= ";" | "," | "#"...\n | "#"..."#" | \n

The <dataset_listfile> entity is a mechanism for including datasets specified in a file into another dataset specification (which may reside on the command line, or in yet another file).

<dataset_listfile> ::= "@"<pathname>

where the ***contents*** of <pathname> are described by this grammar:

<contents of pathname> ::= <dataset> <end-of-file>

<dataset_listfile> refers to a file that contains a valid <dataset> string. Note: the <dataset_sep> character of choice will most likely be a newline character, but this is not a restriction. This use of the "dataset_listfile" concept is illustrated in this example:

> Example: A file named "/home/phil/magpair" containing the two lines:
> {prog:mdi,level:lev1.8,series:fd_M_96m_01d[5599]}
> hmi.M_lev1[2008.05.01/1d:96m]
> used in a command line like:
> Compare_mags in=@/home/phil/mapgair
> would result in 15 magnetograms from MDI and 15 from HMI from 1 May 2008 in the dataset referenced through the command line keyword "in".

In this way, the "@filename" construct acts like an "include" file – at the top level (e.g. the command line in the example above) @ filename causes the datasets specified within filename to be included into the dataset specified at the top level. At this point, there are checks neither on stack overflow (which could arise if @file1 contains @file2, @file2 contains @file3, …, with a sufficient number of files in the chain) nor on cyclical dependencies (which can cause infinite recursion, eg., @file1 contains @file2, and @file2 contains @file1), so please be careful!

In DSDS, a dataset specified by a series name and series number corresponded to a directory on disk. This directory would contain the data files for all the records in the dataset. A dataset was the basic unit of data in DSDS, whereas in DRMS data is managed on a record by record basis. In SUMS, on the other hand, data is handled on the basis of storage units only, and one may choose (when creating a dataseries) to group the data from several records into a single storage unit. Therefore, if the data for one record must be read from tape, for example, then all the data in the same storage unit will be staged as well. This need not necessarily concern the DRMS user, but it may be good to keep in mind when designing dataseries. The difference, however, between a DSDS dataset and a storage unit in SUMS bears emphasizing. The data contained in a DSDS dataset was related in some way, usually corresponding to a defined period in time. The data contained in a storage unit, however,

need not have any particular relationship; a storage unit is for storage only, although it may be common for a storage unit to contain a contiguous sequence of data. The contents of any storage unit are created at the same time, after which time it becomes read-only. Therefore the storage units corresponding to a given dataseries may not all have data for the same number of records.

Record Set

A recordset is a set of records from a single series from a single catalog, and may be described by a name appropriate to its catalog. In the description here the term "recordset" is used interchangeably with "recordset name" where the meaning is obvious from the context. Recordsets of differing types coming from different catalogs as described below may be intermixed in a dataset. The first few characters of each recordset will be used to determine the associated catalog. The rule is as follows: if a leading "{" is found it is a non-DRMS-structured recordset[2]. For now, the only string that may follow "{" is "prog:", indicating a DSDS dataset, but other non-DRMS catalogs may be added in the future. A recordset leading character of "/" specifies a "plain file". This is either a regular file, or a directory. A regular file or a directory with a single file implies a record set with one record and one associated file. A directory with multiple files in it implies a record set with multiple records, and one file per record. A directory with an overview.fits file is treated as a DSDS dataset. If the leading character is not "{" and not "/", then the record set is DRMS-structured dataset.

```
<record_set> ::=
          <dsds_record_set> |
          <plainfile_record_set> |
          <drms_record_set>
```

DSDS and plain file (`<plainfile_record_set>`) name rules are described elsewhere.

DRMS Record Set Definition

DRMS recordsets are a subset of a single series and are specified by a series name and a recordset specifier. If no recordset filter is present the record set consists of the entire series[3].

```
<drms_record_set> ::= <seriesname>{<recordset_filter>}

<seriesname> ::= <namespace>.<series_specifier_name>
```

---

[2] '{' and '}' are necessary for non-DRMS catalogs because specifications for datasets of such catalogs may contain characters, such as ',', that are members of `<dataset_sep>`. Without the curly braces, such characters would confuse the parser.

[3] A notable exception to this syntax is the commonly used module show_info. In that particular case at least one set of empty square brackets is required to get the entire series. This is to reduce accidentally large queries. We mention it here because show_info is likely to be the first module a user attempts to run.

```
<namespace> ::= <name>
<series_specifier_name> ::= <name>
```

A series name consists of two parts, the namespace (project) name and the series specifier name. The namespace provides a private set of dataseries that can be read by all but can only be changed by the owner of the namespace name. The owner may be an individual or a group.

A recordset filter identifies an ordered sequence of records. The requested specific sequence is described in a set of bracketed clauses. Each such clause selects desired records from the dataseries given. The use of square brackets to delimit the recordset specification is to be reminiscent of array indices in languages such as C. The records may be specified by "record_lists" or by direct "record_queries" which specify records using an SQL "where" clause. A record_list is a set of records identified by values of the prime index for that series or by absolute record number. A record_list is then a set of primekey_range_sets or recnum_range_sets. Each such list may be a comma separated set of record ranges. There may be up to one primekey_range_set for each keyword in the prime index.

There may be zero or more record_queries. The final recordset is the logical **and** of the record_queries and record_lists present. Some examples will help – see below.

```
<recordset_filter> ::=
       {<recordset_filter>}(<record_query>|
                           <record_list>)

<record_query> ::=  [!<sql_where_clause>!] |
                    [?<sql_where_clause>?]

<record_list> ::= [{<primekey_name>=}<primekey_range_set>] |
                  [:<recnum_range_set>]

<primekey_name> ::= <name>
<recnum_range_set> ::= <index_range_set>
<primekey_range_set> ::= <index_range_set> | <value_range_set>

<index_range_set> ::= ( #^ | #$ | #<integer> |
                        #<Integer>-#{@<Integer>} |
                        #-#<Integer>{@<Integer>} |
                        #<Integer>-#<Integer>{@<Integer>} |
```

```
                          #<Integer>/<Integer>{@<Integer>}
                        ) {,<index_range_set>}

<value_range_set> ::= ( ^ | $ | <value> |
                          <value>-<value>{@<value_increment>} |
                  <value>/<value_increment>{@<value_increment>}
                        ) {,<value_range_set> }
```

Careful analysis of this rule shows that a recordset_filter is a chain of one or more "[…]" clauses. Each of these may be a range of absolute record numbers (recnums) or an SQL where clause or a list of values for one of the primekey components or a list of index values along a primekey axis. For primekey_range_sets, any or all of the elements of the prime index may be used to identify the records. Each prime index keyword gets its own "[…]" clause. If the prime index contains more keywords than are specified, the selection will include ALL matching records. Thus if the series is a set of lat-lon tiles as a function of time and only a range of times is specified then the selection will include all of the lat-lon tiles for each given time. Similarly if a subrange of one or more keywords is specified then the selected set will include only that range for each of the primekeys. That is, a specification acts as a filter limiting the range of records to be selected. An empty recordset_filter selects the entire series. When multiple keywords constitute the prime index the particular keywords used in a recordset_filter must be clear either by direct identification or by context. If primekey_names are not given the primekey_range_sets will be matched to the components of the prime index in the order in which they appear in the series definition of the prime index. If a leading primekey element is not provided and subsequent elements are used without giving their names explicitly, the leading elements must have empty "[]" sections since the names are matched by the order in the primekey definition.

To search on a keyword that is not in the prime index, one must use an SQL query. The first form, with "!", performs the query directly, whereas the second form, with "?", additionally performs the primekey logic and returns only one record for each value of the prime index. That is, the first form will return all versions of the specified records, and the second form will return only the most recent version of each record. Special syntax is required when forming a query using time strings; see the paragraph on time strings below. When creating a dataseries, it is possible to specify keywords, in addition to the primekeys, to be indexed in the database in order to speed up queries. This should be done whenever a non-prime keyword will be frequently searched on. To learn more about SQL queries, please see http://www.postgresql.org/docs/8.3/static/sql.html.

Finally, we get down to a range of records specified by a range of values of a particular prime index component. The record range may be given as a single record, as a "first" and "last" record in an interval, as a "first" record and duration, or for "slotted prime keywords" as a duration. In the case of a true range, not just a single record, a minimum increment may also be specified.

In the simplest case, a record range is a single value. For example, if a dataseries has a single primekey named YEAR, then a recordset_filter that specifies one record might be "[YEAR=1966]" or simply "[1966]". The appropriate use of this construct depends on the

actual keyword data type, which includes integer, floating, and string types. Integer and string types are easy to deal with since they compare exactly with user entered values. Floating types however may not compare exactly depending on the computational history of a particular value that rounds to a particular printed value. If a record is created with a floating type primekey (named FLOATKEY, for example) whose value is 1992993985.2326, then it is likely that a recordset_filter of "[FLOATKEY=1992993985.2326]" will NOT find the record ingested. There are two solutions for dealing with the problem.

First, whenever a floating type is used as a primekey, all queries could use a range to specify the particular record(s) desired. In other words, do not use queries like "[FLOATKEY=1992993985.2326]"; use "[FLOATKEY=1992993985-1992993986]". However, this is a bit inelegant and leads to the need for the second mechanism: slotting. A "slotted prime keyword" has a floating-point data type (float, double, or time), but it is associated with a second (and hidden) keyword whose data type is integer. Every value that the slotted primekey could have is mapped to an integer value of the hidden keyword. In this manner, a user could specify "FLOATKEY=1992993985.2326" and internally this query might get changed to "FLOATKEY_index=1992993985", where FLOATKEY_index is the associated integer prime keyword. Essentially, the floating-point FLOATKEY axis values are put into FLOATKEY_index "slots" of a certain width. The choice of slot width is entirely application specific. If FLOATKEY values represent seconds since an epoch, then perhaps the values in FLOATKEY_index represent the number of one-second slots since the epoch. In this manner, 985.2326 and 985.7842 would map to slot number 985, but 986.3236 would map to slot number 986. In this example, there is a collision – two floating-point numbers map to a single integer. However, if the FLOATKEY values are guaranteed to be sufficiently "far apart" then every FLOATKEY value maps to a unique FLOATKEY_index value. Time slotting can apply to not only "time" prime keywords, but "degree" prime keywords (e.g., the prime keyword could be a longitude), or any other keyword whose values can be represented by a floating point number.

With a slotted primekey, the user can then use floating-point values in value_range_sets. "FLOATKEY=985.2326" is now a legitimate query, because under-the-hood, this query might be changed into "FLOATKEY_index=985". Of course, FLOATKEY_index is a bona fide prime keyword, so the user could just as easily specify "FLOATKEY_index=235" for a value_range_set. And as explained below, it is possible to select records via *time* slotted prime keywords by specifying a duration (e.g., 60s for 60 seconds, or 96m for 96 minutes, etc.), like "FLOATKEY=96m", or simply "96m" if the actual keyword to be searched is implied. Please see more about slotted keywords in the Slotted Axes section to follow.

Any keyword may be of type time. The usual DRMS format for a time string is YYYY.MM.DD_hh:mm:ss.dd_ZONE, where dd represents fractional seconds. Internally, times are stored as the double precision number of seconds since 1977.01.01_00:00:00_TAI. Trailing parts of a time string may be omitted, in which case the missing fields default to obvious values. The ZONE defaults to UTC. Other formats are allowed; for a full specification see http://jsoc/jsocwiki/JsocTimes. (needs updating) If one wants to use a time string in a record_query, they should use the following syntax: $(<time_string>). This is necessary because it will cause the time string to be converted to the internal format before

being submitted to the database. For example, one might form the query "mdi.fd_M_96m_lev18[? T_REC >= $(1996.05.02_TAI) AND T_REC <$(1996.05.03_TAI) ?]".

In addition to specifying single values, a primekey_range_set may contain a range of values. A record interval is a "dash"-separated first and last axis value or axis index value. An interval expressed this way is "closed" on both ends. In other words, if the range is "19-27", then there are 9 records specified. In the case of a primekey of type *time* (NOT a slotted time keyword though) an interval may be a set of records starting at a given axis value or axis index value and continuing "over" an interval of specified duration. An interval expressed this way is closed on the end earliest in time, and open at the other end. For example, if the interval is "2007.12.25_00:00:00-2007.12.25_01:00:00", and observations occur every minute, then there are 60 records specified (2007.12.25_00:00:00, 2007.12.25_00:01:00, 2007.12.25_00:02:00, …, 2007.12.25_00:59:00). The same set of records can be specified by the interval "2007.12.25_00:00:00/1h". (what about other floating-point types?)

If the primekey is a slotted time keyword, then the interval is closed both at its beginning and end. For example, if the interval is "2007.12.25_00:00:00/1m", and time slots are 10 seconds wide, then *7* records are specified (slots 0, 1, 2, 3, 4, 5, 6). For slotted time primekeys, there is one other method for specifying an interval: the beginning time may be specified as an offset from the reference epoch given in the series definition. For example, if the reference epoch were 2007.12.01_00:00:00, then "[24d/1m]" will specify the same interval as "[2007.12.25_00:00:00/1m]" which is also the same as "[2007.12.25_00:00:00 - 2007.12.25_00:01:00]". Likewise, one may specify a single instant in time as an offset from the epoch. This method will also work for floating-point slotted primekeys of other types.

An example of full disk MDI magnetograms in the DRMS catalog for say 17 March 2005 would be:

mdi.fd_M_96m[2005.05.17/1d]

Note here that omitted time parts (hours, minutes, seconds) all default to zero. Then using the axis offset/axis duration method described below we could have:

mdi.fd_V_lev18[3000d/1d]

which would mean the records for the day 3000 in the MDI epoch. In the DSDS catalog this would be specified as:

prog:mdi,level:lev1.8,series:fd_V_01h[72000-72023]

The epoch default is defined in the series definition (see below). In the DSDS case the above spec expands to 24 datasets of 60 records each. In the DRMS case it expands to 1440 records. Of course the DRMS spec could just as well have been:

mdi.fd_V_lev18[72000h/24h]

The optional increment defines a minimum step to allow undersampling of the target data series. Note this is unwise for oscillation studies which are observed with critical sampling. However a 27-day interval of 96-minute spaced magnetograms from HMI could be expressed as:

hmi.fd_M_lev1[2008.05.01/27d@96m]

or equivalently as

hmi.d_M_lev1[2008.05.01-2008.05.26_22h:24m@96m]

As can be seen in the examples here, there are several ways to specify a particular record. The above examples also hint at implied special knowledge about the type of the primekey values. This issue is discussed below.

Thus individual records may be specified in several ways: by axis value, which is an explicit keyword value along a primekey axis; by specifying an axis offset since the series epoch along the primekey axis; by axis index value, which is an integer specifying the record(s) at the nth position along a primekey axis. These will be described in order below.

Axis Value

An explicit absolute value on a prime index axis may be specified. Some rules apply depending on the variable type of the prime keyword. These are as follows:

1. Integer types. No particular restrictions. The min and max values are restricted to the min and max allowed values for the associated data type. The integer types and limits are:
    a. char -128 127
    b. short -32768 32767
    c. int -2^31 2^31-1
    d. longlong -2^63 2^63-1
2. Floating point types. These are IEEE standard float and double types as 32-bit and 64-bit values. A printing conversion format is provided for all keywords and in the floating case this usually provides a limited precision to show the user. Since the database lookup uses exact bit matches and floating point values printed in limited precision involve rounding, the actual bit value may differ depending on the computation that is used to get a value. Thus whenever floating-type keywords are used as prime keys it may be necessary to specify a range to identify the particular record(s) desired. We advise, however, that all floating type primekeys be slotted. This alleviates the need to specify a range, as discussed above.
3. String types. These are ASCII strings which may contain blanks and other punctuation. If they contain no whitespace, quotation is not needed. Comparisons are case-sensitive.
4. Time. Times are a valid DRMS keyword type. Times are stored internally as doubles and thus are subject to the same warnings as other floating point types. An extra service is provided in the case of times in that DRMS defines both “internal” and “external” representations of times. The external form is the standard JSOC time format (see http://jsoc.stanford.edu/jsocwiki/JsocTimes) such as 2009.01.20_17:00_UT. The internal form is seconds since the JSOC epoch of 1977.01.01_00:00:00.0_TAI which is 1976.12.31_23:59:45.000_UT. Times are by default given in UTC.

One may also specify a range of primekey values to be returned using the ”[<value>-<value>]” notation, even in the case of strings, which are stored in their alphanumeric order. An interval of strings or integers is closed on both ends, but an interval of a floating type,

however, is closed on its beginning and open on its end, as noted above. Note also, however, that if a floating type primekey is slotted, then the true primekey is an integer, and so the interval will be closed on both ends.

Another way to specify a range is to give the beginning of the interval and its length using the "[<value>/<value_increment>]" notation. This is equivalent to the above, except for strings, where <value_increment> is meaningless. For slotted primekeys, one may also use the notation "[<value_increment>/<value_increment>]" where the beginning of the interval is given by an offset from the base given in the series definition. For slotted primekeys of type time, the base will be the reference epoch. Value_increments must obey the following grammar:

```
<value_increment> ::= <integer> | <real> | <time_increment>
<time_increment> ::= <real><time_increment_specifier>
<time_increment_specifier> ::= s | m | h | d
```

Whenever a time_increment specifier is missing, seconds are assumed.

Finally, one may append "@<value_increment>" to any interval to specify a minimum step along a particular primekey axis. This is the same as giving a comma-separated list of values. For example, "[5-10@2]" is equivalent to [5,7,9].

The special values "^" and "$" refer to the smallest (first) and largest (last) value, respectively, of the current primekey. They cannot be used in ranges, however, so they are usually used to select a single record. The use of this notation will be discussed in the next section.

Axis Index Value

Axis index values are intended to be used with integer primekeys. Recall, however, that if a floating-point primekey is slotted, then the floating-point values get mapped to integer values. The integer keyword containing these values becomes the true primekey. Since it is highly advised to use slotting for all floating type primekeys, axis index values are a quite general way to specify records. A recordset_filter of "[#n]" using an axis index value is equivalent to a recordset_filter of "[x]" using the actual value of the primekey according to this formula: x = n * step + base, where step defaults to 1, and base defaults to 0. If desired, these defaults can be overridden in the series definition by defining constant keywords KEY_step and KEY_base where KEY is the name of the keyword to which these values will apply. This is not permitted, however, for slotted primekeys. In that case, the primekey would be KEY_index and step and base would automatically be set to their default values. KEY_index_base and KEY_index_step, while allowed as keywords, will not be used for this purpose.

Hence, by default, x = n for integers. For a slotted floating type primekey, say FLOATKEY, if one uses the "[#n]" notation, it will refer to the actual keyword FLOATKEY_index. The relation between FLOATKEY and FLOATKEY_index will be described in the next section.

If a floating type primekey is not slotted, which is strongly discouraged, then one may not use axis index values. (what about strings?)

(insert examples.)

One may also specify ranges using axis index notation just as with axis value notation. In the recordset_filter "[#<integer>-#<integer>]", either integer may be omitted. If the first, the beginning of the interval is taken to be the smallest axis index present; if the second, the end of the interval is taken to be the largest axis index present. Intervals specified with "/" and "@" behave in exact analogy to the axis value notation.

The special values "^" and "$" refer to the smallest and largest existing index values for the current axis. Hence, for integers and slotted primekeys, "[#^]" is exactly equivalent to "[^]", and "[#$]" is exactly equivalent to "[$]". The only possible difference is in the case of non-slotted floating type primekeys. In this case, the "[#^]" and "[#$]" notation is not allowed. (so is the index a predefined keyword?)

For a series with multiple primekeys, "^" and "$" have a slightly different meaning. The recordset_filter "[#$][#$]" means to find the record(s) with the largest axis index value for the first primekey, then out of those records, find the record(s) with the largest axis index value for the second primekey. If there are only two primekeys, this will result in a single record. Note that a recordset_filter like "[#<max_index_value_1>][#<max_index_value_2]" would not necessarily return any records, but "[#$][#$]" always will as long as a record exists. To find the record(s) with the overall largest axis index value for the second primekey, one could use the recordset_filter "[][#$]". There is no way to use this notation to specify the record(s) with the largest axis index value for the first primekey given the maximum axis index value for the second primekey, but one may find such a recordset using an SQL where clause in a record_query.
(what about [KEYNAME2=#$][KEYNAME1=#$]?)

Slotted Axes

For all slotted prime index keywords several ancillary keywords must be defined along with the prime keyword itself. These keywords must be explicitly described in the series definition (see jsd discussion in the JSOC wiki at http://jsoc.stanford.edu/jsocwiki/Jsd). These ancillary keys must be defined as constant keywords, ie., for each, there can be only one value for the entire series. There are several "types" of slotted keywords supported. The list of types may be extended but at present includes two types of equally spaced time series (TS_EQ and TS_SLOT), general slotted values (SLOT), and Carrington rotation and longitudes (CARR). In all cases the ancillary keyword names are formed by adding a suffix to the base prime keyword name. To define a slotted keyword in the jsd, the "record scope" field must specify one of the slotted keyword types (eg., "ts_eq"). If the jsd parser encounters such a keyword, it will ensure that the required ancillary keywords (like, XXXX_epoch, XXXX_step, and XXXX_unit) are also specified, or parsing will fail. If parsing succeeds, then for each recognized primary keyword, a new "index" keyword is created, with a name XXXX_index. This keyword is of type integer, the values for this

keyword are slot numbers, and is the actual keyword used as the primary keyword in the DRMS database.

For each slotted prime keyword type with a base name of XXXX, the ancillary keywords required to be defined in the jsd are given in the following table:

Type = TS_EQ	
XXXX_epoch	Base epoch for axis. Can be a DRMS time, or a DRMS string (if it refers to a named standard epoch). The epoch will be the center of slot 0.
XXXX_step	Time duration defining the width of each slot. Can be a DRMS float or double, in which case the unit of the duration (eg., seconds, minutes, hours, days, etc.) is specified by XXXX_unit, or can be a DRMS string. If the latter, then the string combines the numerical and unit values (for example, 60s refers to a step of 60 seconds.), and XXXX_unit is not required, or ignored if present.
XXXX_unit	String containing the units of the time duration specified in XXXX_step, provided that XXXX_step is not of type DRMS string. Examples are "secs", "mins", "hours", "days". This keyword may be omitted, in which case a unit of seconds is assumed. (what other strings are allowed? "s"? "seconds"?)
Type = TS_SLOT	
XXXX_epoch	As above, but now epoch marks the beginning of slot 0.
XXXX_step	As above.
XXXX_unit	As above.
XXXX_round	A DRMS float or double giving a number of seconds that represents the uncertainty in the slot boundaries. This can often be set to 0.
Type = SLOT – NOT SURE WHY THERE IS A XXXX_unit.	
XXXX_base	Base reference value for axis; axis_offsets are relative to this value. Can be a DRMS float or double.
XXXX_step	Interval defining the width of each slot. Can be a DRMS float or double, in which case the unit of this number is specified by XXXX_unit, or can be a DRMS string. If the latter, then the string combines the numerical and unit values (for example, 60tribbles), and XXXX_unit is not required, or ignored if present. When calculating slot numbers, the unit of the step is assumed to be the unit of the base, so there is never any utilitarian reason for specifying a step unit.
XXXX_unit	Units of step. Since this is arbitrary (eg., can be "tribbles") not used for calculating slot numbers. This keyword value is only informational. This keyword is not optional.
Type = CARR	
XXXX_step	Degree interval defining the width of each slot. Can be a DRMS float or double, in which case the unit of the interval (eg., degrees, arcminutes, arcseconds, milliarcseconds, radians, microradians, etc.) is specified by XXXX_unit, or can be a DRMS string. If the latter, then the string combines the numerical and unit values (for example, 10ms refers to a step of 10 milliarcseconds.), and XXXX_unit is not required, or ignored if

	present.
XXXX_unit	Units of the degree interval specified in XXXX_step, provided that XXXX_step is not of type DRMS string. Can be a string (eg., “degrees”, “arcmins”, “arcsecs”, “milliarcsecs”, “rads”, “microrads”, etc.). This keyword may be omitted, in which case a unit of degrees is assumed.

Below follows a brief description of each type of slotting, as well as the formulas showing how the XXXX_index keyword is generated using the XXXX keyword and ancillary keywords. For simplicity, we shall assume that all keywords for each type of slotting are given in the same units.

(WILL NEED A DIAGRAM HERE TO SHOW THE DIFFERENCE BETWEEN TS_EQ AND TS_SLOT)

TS_EQ:

This type is intended to be used for timeseries of observables such as Dopplergrams. The XXXX keyword is expected to give the nominal time of observation, but the actual time of observation extends from half a slot width before to half a slot width after this time. Hence,

XXXX_index = (XXXX – XXXX_epoch + XXXX_step/2)/XXXX_step

XXXX_step will therefore correspond to the cadence of observations.

TS_SLOT:

This type is intended to be used for a dataseries where each record corresponds to a timeseries of fixed duration. An example would be 36 day timeseries of spherical harmonic coefficents at a cadence of 1 minute. The formula for XXXX_index now becomes

XXXX_index = (XXXX – XXXX_epoch + XXXX_round/2)/XXXX_step

Now XXXX_step will refer to the duration of the timeseries, and XXXX_round will refer to the cadence within the timeseries. So in the above example, XXXX_step = ‘36days’ and XXXX_round = ‘1min’. Note that XXXX_round usually means the same thing as XXXX_step does for TS_EQ, but it is often safe to set XXXX_round to 0.

SLOT:

XXXX_index = (XXXX – XXXX_base + XXXX_step/2)/XXXX_step

CARR:

XXXX_index = (XXXX – CARR0 + XXXX_step/2)/XXXX_step

For the CARR type of slotted primekey, axis values can be specified in one of two formats: as a Carrington Time, CT:<Carrington Time> (eg., CT:722160.263), or as a Carrington Rotation-Carrington Longitude pair, CRCL:<rotation number>:<degrees> (eg., CRCL:2000:20). Need to modify sscan_time() and sprint_time() in timeio.c to recognize these new formats, and convert the results into TIME. Internally, CARR slotted keywords will have type TIME (DRMS_TYPE_TIME). The defined format for type TIME needs to be expanded to include "CT" and "CRCL". These should operate in a manner analogous to format "UTC". The base time used for slotting is always WSO/MDI Carrtime=0, so there is no need to provide a XXXX_base keyword.

For all slotted axis calculations involving intervals, the mapping to slot number is such that if the beginning of the interval falls within a slot, then the interval contains all of that slot. If the end of the interval falls within a slot, then the interval contains all of that slot.

Standard Epochs for slotted time axes

The XXXX_epoch keyword values (for TS_EQ and TS_SLOT type slotted primekeys only) can be expressed (in the jsd) directly as a time string, or as a predefined standard epoch string:

```
<epoch_keyword_value> ::=   <time_string> | <standard_epoch>
<standard_epoch> ::=
                            "JSOC_EPOCH" |
                            "MDI_EPOCH" |
                            "WSO_EPOCH" |
                            "TAI_EPOCH" |
                            "MJD_EPOCH"
```

Where:

JSOC_EPOCH ➔ 1977.01.01_00h:00m:00s_TAI
MDI_EPOCH ➔1993.01.01_00h:00m:00s_TAI
WSO_EPOCH ➔ 1601.01.01_00h:00m:00s_UT
TAI_EPOCH ➔ 1958.01.01_00h:00m:00s_TAI
MJD_EPOCH ➔ 1858.11.17_00h:00m:00s_UT

Versions

In the JSOC DRMS for dataseries with primekeys defined (almost all series) the always present absolute record number (recnum) is used to track the most recent version of each record. All records with the same primekey values are treated as different versions of the same record. Each time a record is added to a dataseries, recnum is incremented. So for

each set of records with the same primekey values the one with the highest recnum value is the most recent one added and is then the current version.

Since records can be selected in several ways the interaction of the version logic with the selection specification must be understood if the desired results are to be obtained. The basic rule is that the version check is made when at least one primekey filter or a "[? ... ?]" clause is provided. Since each selection clause adds to the record selection filtering in the sense of a logical AND the filtering can be done in any order. Since it is hugely more efficient in the database to first select based on indexed quantities, and the primekeys are always indexed, the selection based on primekeys is done first. After the primekey selection is done, the highest version is selected. Only then are any general record_query clauses executed.

If only recnum_ranges and "[! ... !]" clause are given, then the version checking is not done. This can lead to some unexpected results. Suppose a series has only 2 keywords, A and B with A as the only prime key. Now suppose there are only a few records containing:

Recnum	A	B
1	50	red
2	51	blue
3	51	pink
4	52	white
5	53	blue

The query [50-53] will yield records 1,3,4,5.
The query [? B='blue' ?] returns records 2 and 5 but
The query [][? B='blue' ?] returns only record 5

In general the more the request can be limited by specifying a range of prime keys the better. The performance difference can be dramatic as in 30 minutes vs 30 ms in the case of our expected largest series (2 second cadence for 5 years).

NOTE on record_query "where" clauses

The query in the [! … !] is passed directly to the PostgreSQL processor as the contents following the word "*where*". So, the PostreSQL query rules apply. One thing of note is that for matches to string constants the constant must be enclosed in single quotes as in the example above. The query syntax allowed can be quite complex since a where clause may contain embedded sub-queries. For example the following csh command correctly finds the most recent "PCU" file before the time of the current image (IMAGE_SECS):

```
set PCU_INFO = `show_keys "ds=hmi_ground.test_config_files[? recnum = (select recnum from
hmi_ground.test_config_files where type = 'pcu'  and date <= $IMAGE_SECS order by date desc,
recnum desc limit 1) ?]" -p -q key=date seg=file `
```