

JSOC Data Record Management System

Rasmus Munk Larsen

W.W. Hansen Experimental Physics Laboratory, Annex A210

Stanford University, CA 94305-4085

e-mail: rmunk@quake.stanford.edu

October 17, 2005

Contents

1	JSOC catalog organization	4
1.1	The JSOC data series	4
1.2	The JSOC data record	4
1.2.1	Keywords	4
1.2.2	Links	5
1.2.3	Data segments	6
1.3	The JSOC storage unit	6
1.4	Naming	8
2	Data series specification	9
2.1	Global series information	9
2.2	Keywords	10
2.3	Links	11
2.4	Data segments	12
2.5	JSOC Series Definition, Example 1	13
3	Database representation of JSOC data series	14
3.1	Global tables	14
3.2	Series specific tables	16
3.3	Table schemas and SQL code	16
3.3.1	Global tables	16
3.3.2	Series specific tables	17
4	JSOC pipeline architecture	19
4.1	JSOC pipeline sessions	19
4.2	Concurrency and data integrity	21
5	JSOC User API	21
5.1	Function calls and their semantics	21
5.1.1	Session	21
5.1.2	Records	22
5.1.3	Keywords	23
5.1.4	Links	25
5.1.5	Data Segments	25
5.2	Language bindings	26
5.2.1	C bindings	26
5.2.2	Fortran bindings	26
5.2.3	IDL bindings	26
5.2.4	Matlab bindings	26
A	JSOC Internal functions	27
A.1	JSOC API types	27
A.2	JSOC environment functions	31
A.3	JSOC series functions	31
A.4	JSOC record functions	32
A.5	JSOC keyword functions	32
A.6	JSOC link functions	33
A.7	JSOC data segment functions	33

B JSOC Database Interface Layer (DIL)	33
B.1 DIL initialization and connect functions	33
B.2 DIL data manipulation functions	33
B.3 DIL column types and query result types	34
B.4 DIL query functions	34
B.5 DIL query result functions	35
B.6 DIL transaction functions	35
B.7 DIL sequence functions	35
B.8 DIL type conversion functions	35
B.9 Example program	36
C Acronyms	38

1 JSOC catalog organization

In the following we describe the logical organization of the JSOC Data Record Management System (DRMS), also referred to as the JSOC catalog below, and define a number of terms used to describe data in the JSOC at various levels of abstraction.

1.1 The JSOC data series

Each basic sequence of like data objects, typically “images” or other binary data along with associated meta-data, is called a *data series*. A dataserie consists of a sequence of *data records*. Each data record is the data for one step in “time”. Most but certainly not all dataserie are sequences in time. They can be in principle any list of data objects.

1.2 The JSOC data record

A data record is the basic “atomic unit” of a dataserie, or more precisely: The smallest unit that will be individually registered and available for export from a data series in the JSOC catalog. Most (if not all) access to the JSOC archive by both pipeline processing modules and external data export services will be in terms of data records. In other words, what we informally call the “JSOC catalog” is first and foremost a data record catalog.

Data records are each associated with a *record number* within their dataserie. The record number is an ordinal index number assigned internally by the JSOC to each new record being created. It increments by one for every new record, and so maps to record creation order. The series name and record number together uniquely identifies the data record within the JSOC catalog.

All data records belonging to the same data series have identical logical structure, meaning that they have the same *keywords*, *links*, and *data segments*. These terms will be defined in more detail below. Keywords and links will also be referred to as *meta-data*.

The JSOC Application Programming Interface (API) will provide a set of functions, with bindings to host languages including C, FORTRAN, IDL, and maybe MATLAB, that allow programs to connect to the JSOC environment and retrieve and manipulate data records. The API will contain groups of functions that

- create, read or update data records,
- query the JSOC catalog database to retrieve data records whose keywords satisfy a given condition,
- get and set the contents of keywords, links and data segments,
- create new or modify existing data series.

1.2.1 Keywords

A data record contains zero or more (typically many) named keywords that each map to a value of a simple type such as integer, float or string associated with the record. Keywords are often used to store meta-data describing properties, history and/or context of the main image/observable data stored in the record’s data segments. This is a concept familiar from standard file-based data formats, such as FITS, where the FITS header keywords would correspond to the JSOC keywords and the primary binary arrays or tables would correspond to the JSOC data segments.

In the JSOC catalog keywords values will be stored in database tables separate from the files holding the data segments. This makes it possible to

- modify keyword values without having to locate, access and possibly rewrite files on disk or tape,
- rapidly finding data records whose keywords satisfy a given condition by executing a database query,

- rapidly extract time series of the values of keywords from all or a subset of records in a series. This can be useful for, e.g., trend analysis or time series analysis of global properties, e.g. mean value or other image statistics, of data products.

There will be one database table for each series containing the values of keywords and links for all data record in the series. The values for a single data record will be contained in a single row in that table.

Primary index

For many series a *primary index* associated with the principal axis (e.g. time or (latitude, longitude)) associated with the image in the data record is desired. The intention is that the primary index maps to a unique slot on the principal axis for which might exist in multiple versions of the “same observation” (e.g. newer versions could be created to include missing data or fix a bad calibration). Therefore the primary index does not uniquely identify a data record.

The primary index consists of one or more keyword values that are concatenated to form the full index. **[i.e. we should support queries for intervals like the existing [1000-1010] as well as multi-dimensional primary indices, e.g. (time, latitude, longitude) as [1000-1010, 1-20, 5-50] or something like that.]** If two records have keywords values that differ on any of the keywords comprising the primary index, they are considered different data record (w.r.t. the primary index), otherwise they are considered only different versions of the same data record (w.r.t. the primary index).

The default behavior of the JSOC should be to return the most recent version of a data record for a given primary index. Since record numbers are assigned in order of creation the most recent version is record with the highest record number. The primary index has two crucial uses in the JSOC:

1. It allows users to reference data records by their primary index, which will generally have some physical meaning (e.g. for a time series it could be the number of seconds or hours since some epoch). This will also allow programs and scripts to step through datasets in logical (e.g. time) order, rather than in record creation order as given by the record number which is arbitrary.
2. It allows the JSOC database system to maintain column indexes on the keywords corresponding to the primary index of a series. This will vastly speed up queries that select sets of records based on the primary index (possibly in combination with other criteria), and this is probably majority of all queries in the system.

1.2.2 Links

A data record contains zero or more named links. Links are pointers between data records and make it possible for data records to inherit keyword values from each other, and to capture other dependencies between them such as processing history. For example, a data record can contain links to the data records that were used in creating it, such as a dopplergram data record pointing to the filtergrams from which it was created. Links come in two varieties, *static* and *dynamic*:

- A static link points to a specific data record in the target series identified by (target series name, record number).
- A dynamic link is represented by (target series name, primary index value) and points to the latest version among the data records with the specified value of the primary index in the target series. The JSOC resolves/binds the link to the record number of latest version at the time whenever the data record containing the link is opened.

Notice: This corresponds to a simple query link. I think the most efficient way to implement it in Oracle is using two steps. The first step creates a temporary table with just the subset of records we are interested in. The second step extracts just the records with the highest version for each principal index:

```

create or replace view tmpview as
    select * from <series> where <primary index>=<value in link>
with read only;

select *
from tmpview, (select max(<record number>) as tmp_recnum
               from tmpview
               group by <primary index>
               )
where <record number>=tmp_recnum;

```

This form also works for extracting the newest version of all records satisfying a more general condition, just replace “<primary index>=<value in link>” in the first step with a general query condition.

1.2.3 Data segments

A data record contains zero or more named data segments. The data segments contain data of large volume associated with the data record. The contents of the data segments for a given data record is stored in files in a directory on disk (possible on tape) As described below, to make transfer and storage of JSOC data more manageable and efficient, the data segments for multiple data records are grouped together and managed as a single storage unit. This typically gives rise to a directory structure like

```

/PDS1/DU12342/hmi_lev1_fd_V-34236-0000.fits
      hmi_lev1_fd_V-34236-0001.fitz
      hmi_lev1_fd_V-34236-0002.raw
      hmi_lev1_fd_V-34237-0000.fits
      hmi_lev1_fd_V-34237-0001.fitz
      hmi_lev1_fd_V-34237-0002.raw
      hmi_lev1_fd_V-34238-0000.fits
      hmi_lev1_fd_V-34238-0001.fitz
      hmi_lev1_fd_V-34238-0002.raw

```

where in this example the records are assumed to contain three segments stored in .fits, .fitz and .raw files. In general the file name for each data segment could be of the form:

```
<disk>/<unit>/<series>-<record#>-<segment#>.<extension>.
```

The JSOC API provides a set of functions to access and manipulate the contents of the data segments as n -dimensional arrays, hiding the underlying storage format from the user. The API could also contain functions returning the path to files containing data segments, such that the user can manipulate the files directly, with standard software, if she is so inclined [**at her own risk**].

1.3 The JSOC storage unit

The atomic unit of data that is managed by the JSOC storage system is called a *storage unit*. The JSOC storage system is therefore denoted Storage Unit Management System (SUMS). Each storage unit contains the data segment part of datarecords from a single dataserie, and corresponds to the contents of a single directory, [**possibly with subdirectories for each datarecord**]. A *storage unit index* (denoted DSIndex for historical reasons) is stored with each data record and identifies the storage unit holding the data segments for the record.

A storage unit may be stored online on magnetic disk, offline e.g. on a magnetic tape in a cabinet, or nearline on a tape in a robotic tape library. (The particular storage media is not important to the concept). In response to a user’s request to access a particular datarecord the JSOC catalog will identify the storage unit containing that datarecord by looking up its DSIndex. The DSIndex

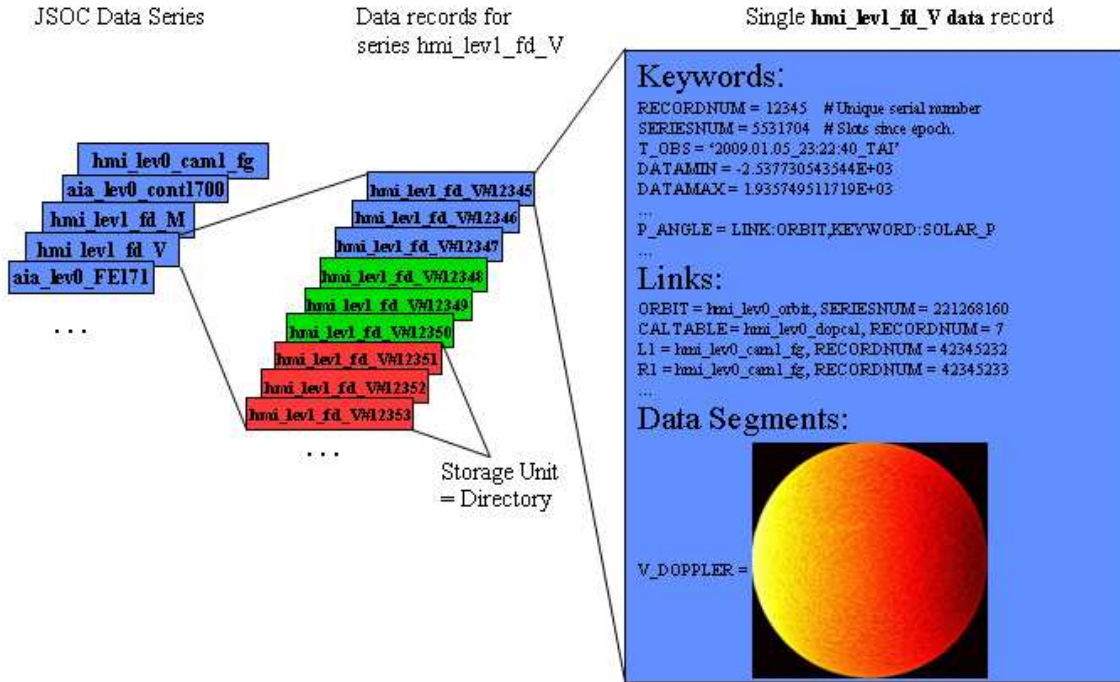


Figure 1: Logical structure of a JSOC data series.

is an index into the SUMS internal catalog which tracks the location of each storage unit. If the requested storage unit is not online the SUMS will allocate storage space, name a directory, and copy the storage unit into that directory. The SUMS will report the working directory pathname to the JSOC catalog where it is accessible to the user. All storage units are owned and managed by the SUMS.

Storage unit are “write-once” objects and clients of SUMS can only perform two operations on them: 1) open existing unit as read-only, 2) create new unit. Deletion or modification of storage units will be restricted to SUMS administrative programs and will require special privileges.

The data records from a particular dataset series will in general be stored in many storage units. The default size of a unit is specified when a series is created. It should be chosen based on knowledge of the size of the data records and how they are likely to be computed, such that a storage unit corresponds to the output of a “natural” processing batch and/or is a convenient size to handle for data export and efficient transfer to tape (the tape archival service can probably further bundle multiple units, if available, together in a single `tar` command to gain further efficiency).

One of the goals of the JSOC data model is allowing user to make a new version of a data record when the value of one or more keywords change without having to copy the large files making up the data segments. This is accomplished by allowing multiple data record to point to the same storage unit. A small example will illustrate this: Consider a simple data series where records have two keywords “seriesnum” (which is the primary index) and “x” and a single data segment “data”. Assume that each storage unit contains a single data record. Let us consider the following sequence of events:

1. new record: create new record with `recordnum=0`, `seriesnum=1`, `x=10.0`, store “data” in new unit
2. keyword change: create new version of record with `recordnum=1`, `x=10.1`
3. data segment change: create new version of record with `recordnum=2`, store updated data in new unit

Here is what happens in each step:

- The first step creates a new entry in the DRMS database holding the keyword values. SUMS creates a new directory and inserts a new storage unit entry into its database. The file containing the data segment is stored in the new directory.

- The second step only modifies the value of a keyword and gives rise to a new entry in the DRMS database. The data segment part is unchanged and the DSIndex refers to the data stored in step 1.
- In the final the data segment is modified and this gives rise to new entries in both the DRMS and SUMS databases, i.e. a new data record and a new storage unit.

The three records are all different version of the same object since they correspond to the same value for the primary index (seriesnum). The final state of the database tables would look something like this:

Data record table				Storage unit table	
recordnum	seriesnum	x	DSindex	DSindex	Path
0	1	10.0	0	0	/PDS/DS00000
1	1	10.1	0	1	/PDS/DS00001
2	1	10.1	1		

[Question from Jesper Schou: Is the example above the only case where records are allowed to point to the same storage units? Should it be allowed that records with different primary index or even records from different series point to the same storage unit? It sort of wrecks the whole data concept. Are there examples where it would be really useful in a way that cannot be accomplished with links?]

1.4 Naming

[This is straight out of Phil's document. I'm not excited about it.]

In the MDI system dataset names were constructed as a 5-part name consisting of a project or program name, a reduction level name, a series name, a series index number, and a version number. The series name contained user readable indicators of the blocking of time into datasets and standardized series numbers were referenced to a common epoch (1 January 1993, 0 UT). Thus the dataset name for hour 0 of full disk velocity images for July 14 1996 is:

```
prog:mdi,level:lev1.5,series:fd_V_01h[30960]
```

After requesting the location of that dataset by a call to a service (e.g. the "peq" program) the user could learn that the dataset is now in the directory "/PDS20/D6362810". If that dataset is not used for a few weeks its online copy will be deleted so the next time it is again requested it will be staged to disk and will appear in another directory. But the user never needs to see the actual storage working directory since the user refers to the data by its descriptive name. The velocity image for say the 10th minute of that hour will be in a file (e.g. 0009.fits) in that working directory. Again the actual file name is not seen by the user since she has used an SSSC provided API function to open the file for the requested minute number.

The JSOC naming system is derived from the MDI system but has some key differences. The concept of a multipart name is eliminated to reflect the MDI actual experience. A new top level name will identify the JSOC data server as a whole to allow a common format for access to e.g. the existing MDI data. The existing name parts simply become user viewable parts of the dataserie name. Thus the same dataset in the example above might be:

```
jsoc:/mdi_fd_V_lev1.5/[t_rec>=1996.07.14_00,t_rec<1996.07.14_01]
```

or

```
jsoc:/mdi_fd_V_01h_lev1.5/[sn=30960]
```

where the syntax here is still TBR.

When the user asks for this dataset in the JSOC system a function provided by the JSOC catalog API will generate a list of all the datarecords in the dataset. That list will include the

working directory (storage unit) and file name and optionally slice within a file for each data record. The user will not normally need to see this information since she will simply use the "open_record" JSOC API function call.

[I don't formally define "dataset" anywhere, it is perhaps just a data record. A "virtual dataset" can be simply a record from a series where the records only contain a set of links and no data segments.]

2 Data series specification

Each data series in the JSOC catalog is defined in a JSOC Series Definition (JSD) file (usually a text file with extension `.jsd`). The JSD contains a description of the global properties of the series, such as its name, owners, date of creation, storage unit size, etc. as well as a concise description of all the keywords, links and data segments that all data records belonging to the series will contain. The JSD is divided into 4 sections, global, keywords, links, and data segments. The syntax of each section is described below.

A new data series is added to the JSOC catalog by parsing the JSD and creating SQL code that when executed by the JSOC catalog database server will create the necessary tables and global table entries to represent the series described in the JSD (see section 3). This can either be done by passing a JSD file to a command line utility

```
command prompt> jsoc_series -create newseries.jsd
```

or by calling the functions

```
JSOC_Series_t *jsoc_parse_description(JSOC_Env_t *env, char *jsd)
int jsoc_create_series(JSOC_Env_t *env, JSOC_Series_t *series)
```

from within a process running in the JSOC environment with the text of the JSD contained in the string argument `jsd`. Since the series name must be unique, the above commands will fail if a series with the same name as specified in the JSD already exists. It is possible to update the definition of an existing series by editing the JSD and either using the command line utility

```
command prompt> jsoc_series -update oldseries.jsd
```

or by calling the functions

```
JSOC_Series_t *jsoc_parse_description(JSOC_Env_t *env, char *jsd)
int jsoc_update_series(JSOC_Env_t *env, JSOC_Series_t *series)
```

Updating a series by adding or removing keywords or links will result in a rebuild of the database tables holding the keyword and link values for the data records. This can be slow for series with many existing records.

2.1 Global series information

This section contains global information about the JSOC data series, that applies to all records and storage units belonging to the series. It takes the form:

```
Seriesname:    <series name>
Description:   <description>
Author:        <author name>
Owners:        <owners>
Unit size:     <storage unit size>
Retention:     <default retention time>
Archive:       <archive flag>
Tape group:    <tape group>
Primary Index: <primary keyword list>
```

where

<series name> is a string containing the name of the JSOC dataserie defined in this file. The series name has to be unique, i.e. there can be no two data series in the JSOC database with the same name.

<description> is a string containing a description of the data series.

<author name> is a string containing the name of the author who created the data series.

<owners> is a string containing a comma separated list of [**Database or Unix?**] usernames with permission to write, delete and modify this series and records and storage units belonging to it.

<storage unit size> is a non-zero integer indicating the number of data records that go into forming a storage unit for this series.

<default retention time> is a string containing either **permanent** or a time interval on the form YYMMDDHH (i.e. 100 for one day, or 1000000 for one year) [**should perhaps choose more standard syntax?**] denoting the default online retention time, i.e. the time for which the files holding the data of the data segments will be kept online. The default retention time can be overridden on a per storage unit basis.

<archive flag> is either 0, meaning that records from this series are not archived on permanent media, or 1 meaning that records from this series are archived to tape. If the archive flag is 0 then the **<tape group>** field is ignored.

<tape group> is an integer identifying the tape group to which this data series belongs. This is used to group together data units belonging to related data series when storing them on tape.

<primary keyword list> A list of zero or more keyword names constituting the primary index of the series. These keywords must be declared in the keyword section of the JSD for it to be valid. If no keywords are given, the record number is implicitly used as primary index.

2.2 Keywords

The keyword section contains declarations of all keywords present in records belonging to the series. Each keyword description consist of "Keyword:" followed by a comma separated list of attributes describing the keyword. Tab or newline characters are ignored. If the keyword is a simple value this takes the following form:

Keyword: **<name>**, **<datatype>**, **<scope>**, **<default value>**, **<format>**, **<unit>**, **<comment>**

where

<name> is a string specifying the name of the keyword.

<scope> can take the value **fixed** which means that the keyword has the same value for all data records, or **<scope>** can take the value **variable** which means that the keyword does not have the same value for all data records.

<datatype> specifies the data type of the keyword value. The following types are recognized: **char**, **short**, **int**, **long**, **long long**, **float**, **double**, **datetime**, **timestamp** and **string**.

<default value> is the default value assigned to this keyword. Default is zero for numerical types, the empty string for string types and "1970-01-01 00:00:00" for date and time types. A fixed keyword has the default value for all records.

<format> is a format string as used by the printf family of functions for formatting an object of type **<datatype>**.

<unit> is a string specifying the physical unit of the keyword value.

<comment> is a string describing the meaning of the keyword.

For example, to declare a keyword containing a floating point value describing velocity associated with a data record, the keyword section would contain a line like

```
Keyword: "v", float, variable, 0.0f, "%12g", "m/s", "velocity", 0
```

A keyword can also be a link pointing to a keyword in a record from a different series, hereby making it possible for data records to inherit keyword values from each other. A link keyword has <datatype> equal to `link` and takes the form

```
Keyword: <name>, link, <linkname>, <target keyword>
```

where

<name> is a string specifying the name of the keyword.

<linkname> specifies the name of the link, which must be defined elsewhere in the same JSD.

<target keyword> specifies the name of the target from from which to get the value in the record pointed to by the link.

For example:

```
Keyword: Tilt, link, HMI_ORBIT, P_Angle
```

This definition means that to get the value of the keyword `Tilt`, follow the link named `HMI_ORBIT` to the data record it points to and get the value of keyword named `P_Angle` from that data record.

[Should we include MIN, MAX and MISSING values to conform with VO tables?]

2.3 Links

Links contain pointers from data records of the data series being defined in the JSD to other data record. Links make it possible for data records to inherit keyword values from each other, and to capture other dependencies between data records such as processing history. For example a data record can contain links to the data records that were used in creating it, such as a dopplergram data record pointing to the filtergrams from which it was created. Links come in two varieties, *static links* and *dynamic links*, and are declared thus:

```
Link: <name>, <target series>, <type>
```

where

<name> is string giving the symbolic name of the link.

<target series> is a string containing the name of the series from which come the record pointed to by the link.

<type> can either take the value `static` which indicates that the link is static, i.e. based on record number, or <type> can type the value `dynamic` which indicates that the link is dynamic, i.e. based on primary index.

A static link points to a single data record from the target series. The link must be established/bound by specifying the record number of the record pointed to. This is done by calling the JSOC function

```
int jsoc_set_link_static(JSOC_DataRecord_t *record, char *linkname, int recnum)
```

from a process running within the JSOC environment with a pointer to the data record passed in as the first argument.

A dynamic link points to the data record from the target series with the highest record number for a given primary index. [If primary indices are simple integers we can use a simple function call as above (or maybe even the same). If primary indices can be composite, we need to be able to pass that into the function.]

2.4 Data segments

The data segment section describes the binary data associated with records from the series being defined in the JSD. Each data segment is an n -dimensional array of a simple type. Various external storage formats of the data segment arrays are available, and are specified by the `<protocol>` field. The general form of a data segment description is

Data: `<name>`, `<form>`, `<scope>`, `<datatype>`, `<naxis>`, `<axis dims>`, `<unit>`, `<protocol>`

The number of fields required depends on the value of `<form>` and `<scope>`, so the following five forms are valid

Data: `<name>`, `generic`, `fixed`

Data: `<name>`, `generic`, `variable`

Data: `<name>`, `array`, `fixed`, `<datatype>`, `<naxis>`, `<axis dims>`, `<unit>`, `<protocol>`

Data: `<name>`, `array`, `variable`, `<datatype>`, `<naxis>`, `<axis dims>`, `<unit>`, `<protocol>`

Data: `<name>`, `array`, `vardim`, `<datatype>`, `<naxis>`, `<unit>`, `<protocol>`

where

`<name>` is a string containing the name of the data segment.

`<form>` is either `generic` which means that the data segment is an unstructured file or `array` which means that the data segment is a multi-dimensional array.

`<scope>` can take the value `fixed` which specifies that the contents of the data segment is the same for all data records in the series. If `<form>` is `generic` then `<scope>` can take the value `variable` which means that a different data segment file is associated with each record. If `<form>` is `array` then `<scope>` can take the value `variable` which means that a different data segment array of fixed size is associated with each record, or `<scope>` can take the value `vardim` which means that a different data segment array of varying size but fixed dimensionality is associated with each record.

`<datatype>` is the data type of the data in the data segment. It can be `char`, `short`, `int`, `long`, `long long`, `float`, `double`, `datetime`, `timestamp` or `string`. Not all file formats support all these data types, see discussion for `<protocol>` below. **[SHOULD ADD FIXED LENGTH STRINGS]**

`<naxis>` is the number of dimensions of an array data segment array, e.g. 1 is a vector, 2 is a matrix, 3 is a cube etc.

`<axis dims>` is a comma separated list of `<naxis>` positive integers giving the dimensions of the data segment array.

`<unit>` is the physical unit of the data segment.

`<protocol>` is the protocol used to store the data segments on disk. Supported protocol are:

FITS Standard FITS format. Supports all data types.

FITZ FITS format compressed losslessly with first differencing plus Rice/Golomb entropy coding. Currently supports 8, 16 and 32 bit integer 1d and 2d arrays.

MSI Multi Scale Image format. Uses a lossless compression algorithm based on wavelet transformation followed by Rice/Golomb entropy coding, storing the bitstream in resolution progressive order. Allows lower resolution (downsampled by powers of two) versions of the image to be extracted rapidly by simply truncating the file. Currently supports 16 bit integer 1d and 2d images with dimensions that are powers of 2.

PNG ?

JPEG ?

TIFF ?

MPEG ?

[probably more formats to come...]

<data segment number> is an integer between 000 and 999. Data segments are numbered according to the order in which they occur in the JSD.

2.5 JSOC Series Definition, Example 1

Below is shown an example of a JSD defining a series named "testclass1". The example has 8 keywords of different types (1 linked and 7 simple), 2 links (one static and one query), and 6 data segments (3 fixed arrays for axis, two main variable data arrays and one generic segment meant for a text file with processing log):

```
##### Global series information #####
Seriesname:      "testclass1"
Description:     "This series is for testing only."
Author:          "Rasmus Munk Larsen"
Owners:          "rmunk"
Unitsize:        10
Archive:         1
Retention:       permanent
Tapegroup:       127
Primary Index:

##### Keywords #####
# Format:
# Keyword: <name>, link, <linkname>, <target keyword name>
# or
# Keyword: <name>, <datatype>, {fixed | variable}, <default value>, <format>, <unit>, <comment>
#
Keyword: "keywd0",      link, "link1", "keywd0"
Keyword: "keywd1",      char, variable, '\0', "%d", "unit1", "Comment1"
Keyword: "keywd2",      int, variable, 0, "%d", "unit2", "Comment2"
Keyword: "keywd3",      float, variable, 0.0f, "%f", "unit3", "Comment3"
Keyword: "keywd4",      double, variable, 0.0, "%lf", "unit4", "Comment4"
Keyword: "keywd5",      datetime, variable, "1970-01-01 00:00:00", "%-s", "unit5", "Comment5"
Keyword: "keywd6",      timestamp, variable, "19700101000000", "%-s", "unit6", "Comment6"
Keyword: "keywd7",      string, variable, "", "%-s", "unit7", "Comment7"

##### Links #####
# Format:
# Link: <name>, <target series>, { static | dynamic }
#
Link: "link0", "testclass0", static
Link: "link1", "testclass0", dynamic

##### Data segments #####
# Data: <name>, <form>, <scope>, <datatype>, <naxis>, <axis dims>, <unit>, <protocol>
#
Data: "x-axis",          array, fixed, float, 1, 100, "m", fitz
Data: "y-axis",          array, fixed, float, 1, 200, "m", fitz
Data: "z-axis",          array, fixed, float, 1, 50, "m", fitz
Data: "pressure",        array, variable, float, 3, 100, 200, 50, "kg/(s^2*m)", fitz
Data: "velocity",        array, variable, float, 4, 100, 200, 50, 3, "m/s", fitz
Data: "processing log",  generic, variable
```

3 Database representation of JSOC data series

[This is a modified version of what I wrote for Jim’s CDRL document, so it repeats (and probably sometimes contradicts) some of what has been said above. It uses a different example than the one in section 2.5]

The data making up the records and units of a data series specified in a JSD are represented in the JSOC by a collection of database tables and records, as well as files and directories on disk and/or tape. The information in the JSD is stored in a number of global tables, while the data representing the values of keywords and links for each data record are stored in a single database table specific to the series. Finally, the data representing the data segments of a record are stored in files on disk or tape, managed by the SUMS. In the following we describe the details of these tables.

3.1 Global tables

The JSOC catalog will maintain a set of global tables that define the structure of data records belonging to each series, and contain information shared by all record of the series, such as tape storage group, default online retention time and whether records from the series should be archived to tape.

An example of what these tables might look like is shown in Figure 2.

- The **master_series_table** contains a list of all JSOC data series with descriptions of the series, information about when and by who they were created, in addition to information about the storage policy for the series.
- The **master_keyword_table** contains a list of all keywords for all JSOC data series. Each row in the table describes things like the name of the keyword, the data type, default output format and physical unit of its value and whether the keyword is "inherited" by following a link to another data record or whether it is stored as a simple value in the "series table", which is the main table holding actual the keyword values for all records belonging to it (see below). Additional information might include whether the database should maintain an index on this keyword value. This is done to speed up database queries with conditions involving this keyword value.
- The **master_link_table** contains a list of all links for all JSOC data series. Each link has a name and a target series to which they point, and can be either static, i.e. pointing to a specific data record in the target series defined by (record number, version), or it can be a "query link". A "query-link" is can be represented as (link name, series name, query) tuple. The intention of the later form is that the link can be bound/resolved dynamically by evaluating the query condition either at creation time or when the data record is opened for reading. This can for example be used to create a link that automatically resolves to the most recent data record of a data series, e.g. the most recent version of a calibration table. The syntax used in query links is TBD.
- The **master_data_table** contains a list of the data segments for records belonging to JSOC data series. A data segment is described by its name, type, physical unit, dimensions and the storage protocol used when accessing the contents of the data segment in secondary storage.

The master tables above merely describe the structure of data records belonging to data series and are therefore reasonably small. The number of rows in these tables is proportional to the number of data series [which is probably no more than a few hundreds, maybe a thousand.]. The bulk of the JSOC catalog is a set of tables, one for each series, containing the actual keyword values for all data records. In addition, a special sequence table is maintained for each series. A sequence is a special database table type containing a counter that can be read and incremented in an atomic operation. It is used to guarantee that unique record numbers are generated even when several modules concurrently are creating new records belonging to the same series.

master_series_table:

Series	Author	Created	Description	Archive	Tapegroup	Retention	Owner	Index
hmi_fd_v	Jesper	2006-05-02 10:52:44	Doppler velocity	1	2	40000	production	T_Obs
hmi_lev0_fg	Rasmus	2006-05-02 10:52:42	Filtergram ...	1	1	6000	production	T_Obs
testclass1	Rasmus	2004-10-06 13:14:15	simple testclass	0	0	0	rmunk	Time
...

master_keyword_table:

Series	Keyword	Type	Scope	Default value	Format	Unit	Linkname	Target Name
hmi_fd_v	T_Obs	datetime	variable	'1970-01-01'	"%F %T"	s	NULL	NULL
hmi_fd_v	D_Mean	float	variable	'0.0'	"%12.5f"	g/s	NULL	NULL
...
hmi_fd_v	P_Angle	link	NULL	NULL	NULL	NULL	Orbit	PANGLE
hmi_sht_v	lmax	int	variable	'0'	"%d"		NULL	NULL
...

master_link_table:

Series	Link	Target_Series	Type
hmi_fd_v	Orbit	sdo_fds	dynamic
hmi_fd_v	L1	hmi_fg	static
hmi_fd_v	R1	hmi_fg	static
...
hmi_fd_v	Caltable	hmi_dopcal	static
...

master_data_table:

Series	Name	Protocol	Type	Unit	Naxis	Axis
hmi_fd_v	velocity	FITS	float	m/s	2	4096, 4096
hmi_lev0_fg	intensity	FITZ	short		2	4096, 4096
...

Figure 2: Example illustrating the structure of the global tables in the JSOC catalog.

hmi_fd_v:

ID	...	T_Obs	D_Mean	...	Orbit_ID	L1_ID	DSIndex
0	...	'2008...'	123.456	...	7	2341	123456
1	...	'2008...'	123.456	...	7	2341	123457
2	...	'2008...'	234.567	...	8	2361	123456
...			
9588392	...	'2010...'	234.567	...	48	64112361	123457

hmi_fd_v_seq:

ID
9588392

Figure 3: Illustration of the structure of the series specific tables for the series **hmi_fd_v**. The main record table is named **hmi_fd_v** and the record number sequence counter table is named **hmi_fd_v_seq**.

3.2 Series specific tables

The two tables **hmi_fd_v** and **hmi_fd_v_seq** in Figure 3 illustrate what the main record table and the sequence counter table for the series "hmi_fd_v" might look like:

Each row in the table **hmi_fd_v** represent a data record from the series **hmi_fd_v**. The first two columns contain record number (ID) and version which together with the series name uniquely identifies the record within the JSOC environment. The values of simple keywords (T_Obs and D_Mean in the example) are stored in the table along with record number (ID) and version of objects pointed to by links. For example, the second row in the table represents data record (**hmi_fd_v**, 1, 1) which contains a link called L1 pointing the record (**hmi_fg**, 2341, 1) from the **hmi_fg** series. For this record the query link named Orbit has been resolved to point to the record (**sdo_fds**, 7, 1) from the series **sdo_fds**.

Finally, the "hmi_fd_v" table contains for each record a pointer to the storage unit in which the contents of the data segments of the record is stored. This pointer (here called "DSIndex") is an index into a database table maintained by the SUMS sub-system. DSIndex can be resolved by the SUMS to a path of a directory containing the data files of the data segments.

3.3 Table schemas and SQL code

3.3.1 Global tables

The global tables **master_keyword_table**, **master_keyword_table**, **master_link_table**, and **master_data_table** are created once by the JSOC database administrator (DBA) with the following SQL code:

```

create table mdc_master_series_table
(
    series      varchar(64)    not null,
    author      varchar(64)    not null,
    owners      varchar(64)    not null,
    created     timestamp      not null,
    description text,
    unitsize    integer        not null,

```



```

        archive      integer      not null,
        tapegroup    integer,
        retention    timestamp    not null,
        primary key (series)
);

create table mdc_master_keyword_table
(
    series   varchar(64)      not null, /* Name of series. */
    keyword  varchar(64)      not null, /* Name of keyword. */
    number   integer          not null, /* Keyword number */
    linkname varchar(255),    /* should be NULL for ordinary keywords. */
    type     varchar(20)      not null, /* Data type */
    maxsize  integer          not null, /* Max size of data type */
    format   varchar(20)      not null,
    unit     varchar(16)      not null, /* Physical unit */
    comment  varchar(255)     not null,
    indexed  integer          not null,
    primary key (seriesname, keywordname)
);

create table mdc_master_link_table
(
    series   varchar(64)      not null, /* Name of series. */
    link     varchar(64)      not null, /* Name of link. */
    number   integer          not null, /* Link number */
    target_series varchar(64)  not null,
    type     varchar(20)      not null, /* Simple or query */
    query    varchar(255)     not null, /* Database query */
    binding  varchar(10)      not null, /* Bind time */
    comment  varchar(255)     not null,
    primary key (seriesname, linkname)
);
s
create table mdc_master_data_table
(
    series   varchar(64)      not null, /* Name of series. */
    data     varchar(64)      not null, /* Name of data segment. */
    number   integer          not null, /* Segment number */
    type     varchar(20)      not null, /* Data type */
    unit     varchar(16)      not null, /* Physical unit */
    naxis    integer          not null, /* Rank = number of dimensions */
    axis     varchar(255)     not null, /* comma separated list of axes dimensions */
    comment  varchar(255)     not null,
    primary key (seriesname, dataname)
);

```

3.3.2 Series specific tables

We now continue with the example from Section 2.5. Below is shown the SQL code that will be executed by parsing the JSD of Example 1:

```

/* Register series info in the global table. */
insert into MDC_MASTER_SERIES_TABLE values

```

```

('testclass1', 'Rasmus Munk Larsen', 'rmunk', now(), "This series is for testing only.", 10, 1, 127,

/* Register keywords in the global table. */
insert into MDC_MASTER_KEYWORD_TABLE values
    ('testclass1','keywd0', 0, 'newest', 'link', NULL, NULL, NULL, NULL, NULL);
insert into MDC_MASTER_KEYWORD_TABLE values
    ('testclass1','keywd1', 1, NULL, 'char', 1, '%d', 'unit1','comment1', 0);
insert into MDC_MASTER_KEYWORD_TABLE values
    ('testclass1','keywd2', 2, NULL, 'int', 4, '%d', 'unit2','comment2', 1);
insert into MDC_MASTER_KEYWORD_TABLE values
    ('testclass1','keywd3', 3, NULL, 'float', 4, '%f', 'unit3','comment3', 0);
insert into MDC_MASTER_KEYWORD_TABLE values
    ('testclass1','keywd4', 4, NULL, 'double', 8, '%lf', 'unit4','comment4', 0);
insert into MDC_MASTER_KEYWORD_TABLE values
    ('testclass1','keywd5', 5, NULL, 'datetime', 22, '%-s', 'unit5','comment5', 0);
insert into MDC_MASTER_KEYWORD_TABLE values
    ('testclass1','keywd6', 6, NULL, 'timestamp', 22, '%-s', 'unit6','comment6', 0);
insert into MDC_MASTER_KEYWORD_TABLE values
    ('testclass1','keywd7', 7, NULL, 'string', 4, '%-s', 'unit7','comment7', 0);

/* Register links in the global table. */
insert into MDC_MASTER_LINK_TABLE values
    ('testclass1','link0', 0, 'testclass0', 'static', '', '', '');
insert into MDC_MASTER_LINK_TABLE values
    ('testclass1','newest', 1, 'testclass0', 'query', 'time=(select max(time) from testclass0)', 'creation');

/* Register data segments in the global table. */
insert into MDC_MASTER_DATA_TABLE values
    ('testclass1','x-axis', 0, 'float', 1, '100', 'm', 'fitz');
insert into MDC_MASTER_DATA_TABLE values
    ('testclass1','y-axis', 1, 'float', 1, '200', 'm', 'fitz');
insert into MDC_MASTER_DATA_TABLE values
    ('testclass1','x-axis', 2, 'float', 1, '50', 'm', 'fitz');
insert into MDC_MASTER_DATA_TABLE values
    ('testclass1','pressure', 3, 'float', 3, '100, 200, 50', 'kg/(s^2*m)', 'fitz');
insert into MDC_MASTER_DATA_TABLE values
    ('testclass1','velocity', 4, 'float', 4, '100, 200, 50, 3', 'm/s', 'fitz');

/* Create the main table to hold per-record information. */
create table testclass1
(
    recnum          integer not null, /* Record number */
    ver             integer not null, /* Record version */
    link0_recnum    integer,
    link0_ver       integer,
    newest_recnum    integer,
    newest_ver       integer,
    keywd1          "char"           default '\0',
    keywd2          integer           default 0,
    keywd3          real              default 0.0,
    keywd4          double precision  default 0.0,
    keywd5          varchar(22)       default '1970-01-01 00:00:00',
    keywd6          varchar(22)       default '19700101000000',
    keywd7          text              default ''

```

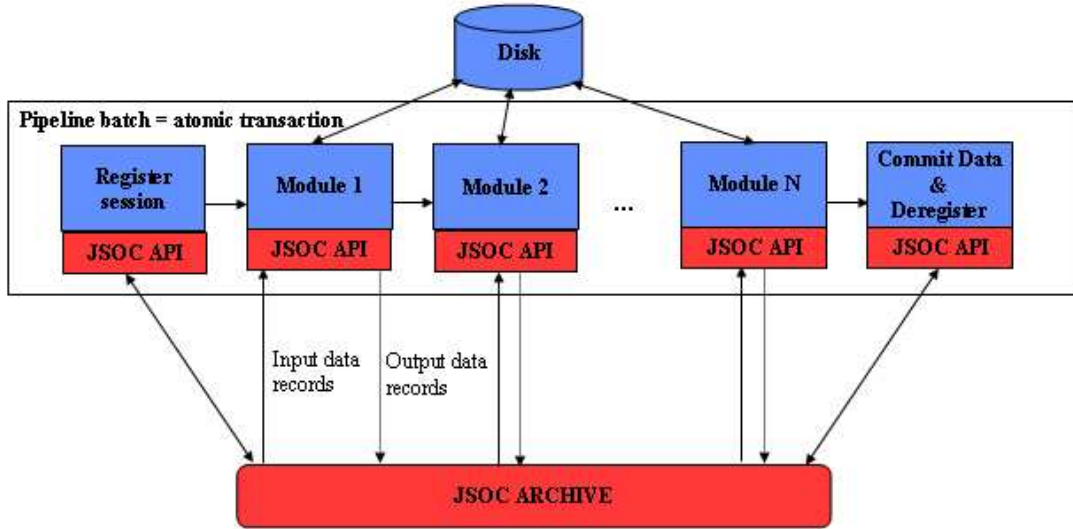


Figure 4: Illustration of a JSOC session.

```

storage_index integer not null unique, /* Index of the record in the main storage table
                                         holding information about where the data
                                         files are stored. */

primary key(recnum,ver)
);
/* Create indexes explicitly requested in the series specification. */
create index keywd1_idx on testclass1(keywd1);

/* Create an atomic counter for creating new record numbers for this series. */
create sequence testclass1_seq;

/* Insert default values as record with (recnum,ver) = (0,0). */
insert into testclass1 values
(0,0,0,0,0,0,'\'0', 0, 0.0, 0.0, '1970-01-01 00:00:00','19700101000000', '');

```

4 JSOC pipeline architecture

4.1 JSOC pipeline sessions

Pipeline processing in the JSOC will occur in batches or *session*, which consist of the execution of a sequence of modules, see Figure 4. A session is an atomic transaction in the sense that either all or none of the data records created in the session are archived. This is achieved by controlling all database access and creation of data records and units in a single DRMS instance (similar to the PE process in MDI), which acts as the session master process.

The master DRMS process begins a new session by opening a connection to the Oracle database server and issuing a **BEGIN TRANSACTION** command. All subsequent database operations (reading, modifying or creating data records) performed by DRMS on behalf on client modules will take place within a single database transaction. At the end of a successful session DRMS will issue a **COMMIT** command and the changes made to the database will become visible to other users of the database. However, all data record created or modified by a module *will* be immediately visible to subsequent modules executed within the same session.

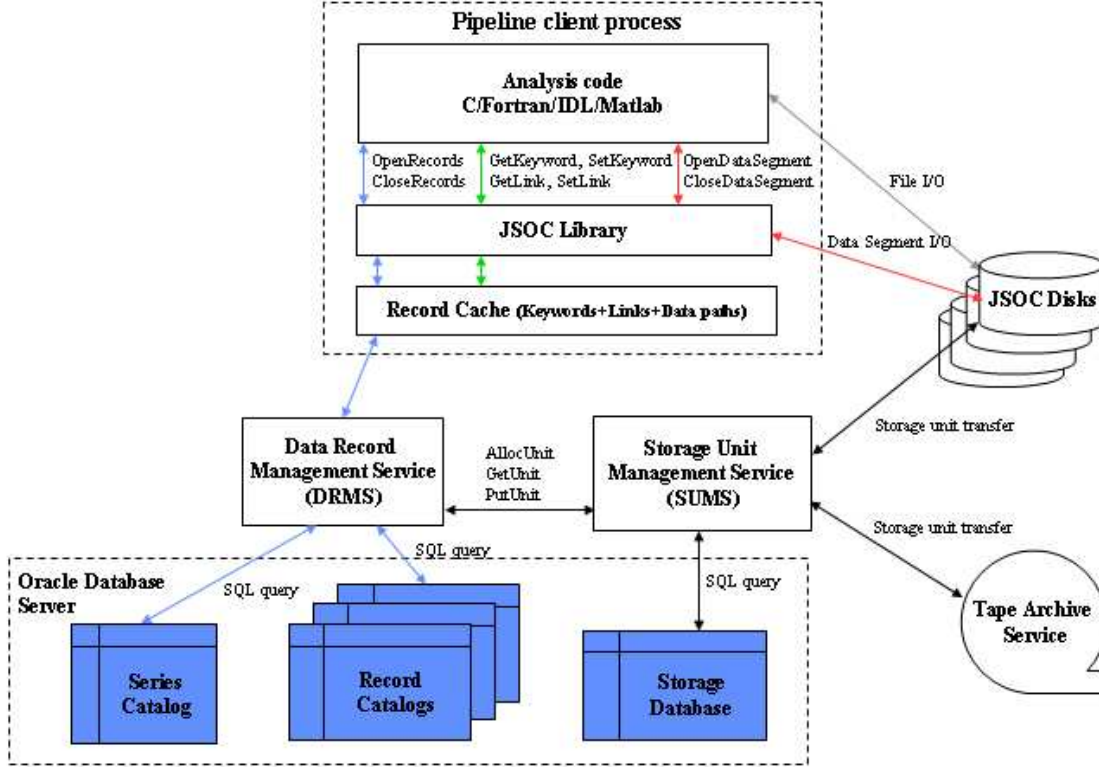


Figure 5: Client-server architecture of JSOC pipeline.

If an error occurs or one of the client modules finishes with an abort request the DRMS will issue a ROLLBACK command and any changes made to the database will be undone, such that to other users the database will appear as if the session never took place.

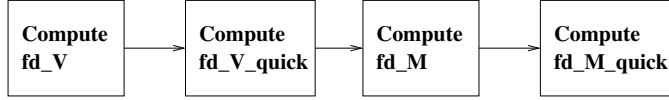
The master DRMS process keeps track of all storage units being read and created within a session in an in-memory table.

When an existing data record is opened for reading DRMS will query the database determine which storage unit it belongs to and ask SUMS for a path to the directory where that unit resides (possibly the unit has to be read from tape first). DRMS will then insert an entry in its internal table stating that the unit storage unit is open for reading.

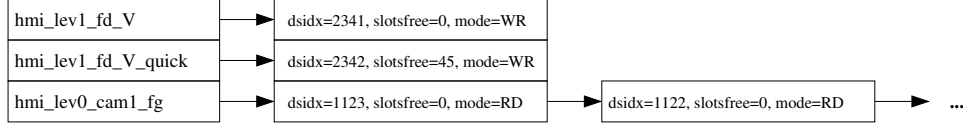
When a new data record is created DRMS checks its internal table to see if it has any data units belonging to the same series open for creation which still have empty slots left. If so it assigns an empty slot to the data record. If not it asks SUMS to allocate a new data unit for the series and inserts an entry for it into its internal table.

In Figure 6 is an illustration of what the internal table in DRMS might look like during the execution of an imaginary pipeline session containing four modules executed sequentially. The two first modules read data records from the series `hmi_lev0_cam1_fg` and write records to the series `hmi_lev1_fd_V` and `hmi_lev1_fd_V_quick` respectively. The two last modules read data records from the series `hmi_lev0_cam2_fg` and write records to the series `hmi_lev1_fd_M` and `hmi_lev1_fd_M_quick` respectively. It is assumed that the unit size for `hmi_lev1_fd_V` and `hmi_lev1_fd_V_quick` is 90 (would correspond to one hour of data at a 40 second cadence).

Example pipeline session:



DRMS state during half-way through execution of "Compute fd_V_quick":



DRMS state at the end of the session:



Figure 6: DRMS state during session.

4.2 Concurrency and data integrity

5 JSOC User API

The JSOC will provide a software API that allows pipeline programs written in C, Fortran, IDL or Matlab to insert new data record, read and update the values of keywords and links and access the contents of the data segments.

5.1 Function calls and their semantics

5.1.1 Session

`DRMS_Session_t * drms_handle = drms_connect(char *DRMS_locator, char *user, char *passwd)`

Called by the client to open a new socket to a running DRMS instance. The DRMS process spawns (and detaches?) a new thread which will pass database calls from the new client through to the Oracle database connection shared by all clients of the same DRMS instance. The string "DRMS_locator" is some TBD descriptor that specifies which DRMS instance to connect to and how. It could be of the form "hostname:port". If a connection cannot be established the `drms_handle` returned is NULL.

`void drms_disconnect(DRMS_Session_t *drms_handle, int abort)`

Called by the client to disconnect from DRMS. If `abort` is 1 the client issues an abort command to the DRMS instance which will roll back all changes made by clients in the current session. If `abort` is 0 then all data records created or modified by the client will be committed to the archive unless a subsequent module issues an abort.

```
int status = drms_commit(DRMS_Session_t *drms_handle)
```

Called by the client to force DRMS to commit the data records created so far in the session. DRMS will commit all records inserted into the database and tell SUMS to archive data units created so far. This call can be used for creating check-points in sessions.

```
void drms_abort(DRMS_Session_t *drms_handle)
```

`drms_abort` first calls `drms_disconnect` with `abort=1` to discard all data records and storage units created in the current session. It then prints to `stderr` an error message containing the file and line number where the call to `drms_abort` occurred, followed by a stack dump. Finally the program will be terminated with a non-zero exit code.

5.1.2 Records

```
int status = drms_open_records(DRMS_Session_t *drms_handle, char *dataset_desc, int *num_records,
DRMS_Record_t ***records, int mode)
```

Open existing data records. The parameter `dataset_desc` contains a dataset descriptor string using the syntax described in Section 1.4. The number of records matching the descriptor is returned in `(*num_records)`, and an array containing pointers to the records retrieved is returned in `(*records)`. If `mode=READONLY` the record will be opened as read-only. Opening with `mode=CLONE_COPY` is shorthand for opening and cloning (with `mode=COPY_DATA`) records in a single call and corresponds to the following code snippet:

```
int num_recs, *flags;
DRMS_Record_t **old_recs, **new_recs;
...
drms_open_records(drms_handle, dataset_desc, &num_recs, &old_recs, READONLY);
new_recs = (DRMS_Record_t **) malloc(num_records*sizeof(DRMS_Record_t *));
mode = malloc(num_records*sizeof(int));
for (i=0; i<num_recs; i++)
    flags[i] = COPY_DATA;
drms_clone_records(drms_handle, num_records, old_recs, new_recs, flags);
for (i=0; i<num_recs; i++)
    flags[i] = DISCARD;
drms_close_records(drms_handle, num_recs, old_recs, flags);
```

Opening with `mode=CLONE_SHARE` is shorthand for opening and cloning with `mode=SHARE_DATA`. See the description of `drms_clone_records` below.

```
int status = drms_clone_records(DRMS_Session_t *drms_handle, int num_records, DRMS_Record_t
*input[num_records], DRMS_Record_t *output[num_records], int mode[num_records])
```

Clones a set of existing data records. For each input record `input[i]`, $i = 0, \dots, \text{num_records} - 1$, create a copy of it, assign the copy a new unique record number.

The parameter `mode` can take the values `COPY_DATA`, `SHARE_DATA` and determines whether the data segment files are copied or the old data segment shared between the old and the new record. If `mode=COPY_DATA` then the data segments are copied to a new slot in a storage unit and the caller will have permission to open them for writing. If `mode=SHARE_DATA` the new record will point to the data unit slot containing the data segments of the old record, and will not have permission to open them for writing. **[exactly how permissions on datasegments will work is TBD]**

If the value returned in `status` is 0 the call succeeded. If `status` is -1 an unknown error occurred. **TBD: [Add positive status codes for known error conditions]**

```
int status = drms_create_records(DRMS_Session_t *drms_handle, char *series, int num_records,
DRMS_Record_t *records[num_records])
```

Create `num_records` new records from the series whose name is given by the string argument `series` and return handles to the records in the array `records`. Each record will be given a unique record number and its keywords and links will be initialized to their default values. If the series definition contains one or more data segments, DRMS will assign a storage unit to each record. This may involve allocate new data units from SUMS if not enough slots available in new storage units already opened for writing for the series. If the value returned in `status` is 0 the call succeeded. If `status` is -1 an unknown error occurred. **TBD: [Add positive status codes for known error conditions]**

```
int status = drms_close_records(DRMS_Session_t *drms_handle, int num_records, DRMS_Record_t
*records[num_records], int action[num_records])
```

This call tells DRMS to close `num_records` data records identified by the handles in the array argument `records`. The optional array `action` has an entry for every record telling DRMS what to do with the record. Entries in `action` can take the values COMMIT and DISCARD.

A value of `action[i]=COMMIT` will tell DRMS to insert the keyword and link values for record *i* in the record database, and to mark any of the data segment files written to a storage unit for subsequent archiving by the SUMS.

A value of `action[i]=DISCARD` will tell DRMS to discard the record, i.e. not insert it into the database. DRMS will also delete data segment files associated with the record, unless the record was created by `drms_clone_records` with `mode=SHARE_DATA`.

For “old” records opened with `drms_open_records` in read-only mode, `action` always defaults to DISCARD. For “new” records created by calling

- `drms_open_records` with `mode=CLONE_COPY` or `mode=CLONE_SHARE`,
- `drms_clone_records` with `mode=COPY_DATA` or `mode=SHARE_DATA`
- `drms_create_records`

the value of `action` always defaults to COMMIT. If `action` is NULL, the defaults just mentioned apply. `drms_clone_records` frees all memory associated with the records.

If the value returned in `status` is 0 the call succeeded. If `status` is -1 an unknown error occurred. **TBD: [Add positive status codes for known error conditions]**

```
char * drms_get_record_path(DRMS_Record_t *record)
```

Returns a malloced string containing a path to the directory where data segment files associated with `record` reside. It is the responsibility of the caller to free the returned string.

```
void drms_free_record(DRMS_Session_t
drms_handle, DRMS_Record_t *record)
```

Frees the memory associated with `record`.

```
void drms_free_record(DRMS_Session_t *drms_handle, int num_records,
DRMS_Record_t *records[num_records])
```

Frees the memory associated with `record[i]`, *i*=1,...,*num_records*-1.

5.1.3 Keywords

```
int status = drms_setkey_char(DRMS_Record_t *record, char *keyname, char value)
```

```
int status = drms_setkey_int(DRMS_Record_t *record, char *keyname, int value)
```

```
int status = drms_setkey_short(DRMS_Record_t *record, char *keyname, short value)
```

```
int status = drms_setkey_longlong(DRMS_Record_t *record, char *keyname, long long value)
```

Table 1: List of status codes returned by `drms_setkey` and `drms_getkey` family of functions.

status	C macro	Meaning
0	SUCCESS	Success, no loss of accuracy
1	JSOC_INEXACT	Success, possible loss of accuracy
-1	JSOC_RANGE	Failure, value out of range
-2	JSOC_BADSTRING	Failure, trying to convert invalid string to numeric type
-3	JSOC_MISSING	Failure, <code>keyname</code> refers to non-existing keyword

Table 2: Matrix of `status` return values from the `drms_setkey` and `drms_getkey` family of functions depending on the types of source and destination operands in the call. In addition all calls can return -3. See Table 1 for the meaning of the various values.

destination type source type	char	short	int	long long	float	double	string
char	0	0	0	0	0	0	0
short	-1, 0	0	0	0	0	0	0
int	-1, 0	-1, 0	0	0	0, 1	0	0
long long	-1, 0	-1, 0	-1, 0	0	0, 1	0, 1	0
float	-1, 0, 1	-1, 0, 1	-1, 0, 1	-1, 0, 1	0	0	1
double	-1, 0, 1	-1, 0, 1	-1, 0, 1	-1, 0, 1	-1, 0, 1	0	1
string	-2, -1, 0	-2, -1, 0	-2, -1, 0	-2, -1, 0	-2, -1, 1	-2, -1, 1	0

```
int status = drms_setkey_float(DRMS_Record_t *record, char *keyname, float value)
int status = drms_setkey_double(DRMS_Record_t *record, char *keyname, double value)
int status = drms_setkey_string(DRMS_Record_t *record, char *keyname, char *value)
```

These functions are used to assign a value (given by the argument `value`) to a keyword of the data record pointed to by `record`. The name of the keyword is given by the string argument `keyname`.

When the type of `value` and the named keyword agrees the assignment always succeeds and `status=0` is returned. List of possible return value for `status`:

Notice that the string to floating point conversions never return `status=0`, but instead return `status=1` for a successful conversion, indicating that accuracy may have been lost. Certain strings like "0.25" can in principle be converted to floating point without loss of information, whereas strings like "0.1" do not have a finite binary representation. The C library functions `strtof` and `strtod` used internally in JSOC do not provide information to distinguish the two cases, so we take the more cautious approach of always returning `status=1` to remind the user that rounding may have occurred. The same argument holds for conversion from floating point to string.

```
char drms_getkey_char(DRMS_Record_t *record, char *keyname, int *status)
short drms_getkey_short(DRMS_Record_t *record, char *keyname, int *status)
int drms_getkey_int(DRMS_Record_t *record, char *keyname, int *status)
long long drms_getkey_longlong(DRMS_Record_t *record, char *keyname, int *status)
float drms_getkey_float(DRMS_Record_t *record, char *keyname, int *status)
double drms_getkey_double(DRMS_Record_t *record, char *keyname, int *status)
```



```
char * drms_getkey_string(DRMS_Record_t *record, char *keyname, int *status)
```

These functions return the value of keyword from the data record pointed to by `record`. The name of the keyword is given by the string argument `keyname`. If `status` is not NULL, one of the exit codes listed in Table 1 will be returned. If `status` is NULL two things can happen: If the status value would have been non-negative the call returns successfully. If the status value would have been negative the `drms_abort` is called and the program is terminated and information about where the error occurred is written to `stderr`.

5.1.4 Links

```
int status = drms_setlink(DRMS_Record_t *record, char *linkname, int index)
```

Set the link named `linkname` in the data record associated with `record` to point to a given record in the target series. The target series is defined in the global series definition of the series to which `record` belongs. If the link type is static, `index` should contain a unique record number in the target series. If the link type is dynamic, `index` should contain a primary index value in the target series.

If `status=0` the call succeeded. If `status=-1` the call failed because there is no link named `linkname` in `record`. **[What to do if index refers to a record in the target series that does not (yet) exist? Should this be checked at runtime by querying the database?]**

```
int status = drms_getlink(DRMS_Record_t *record, char *linkname, char **target_series,  
int *index)
```

Return the target series and index of a link. If a link with the name in `linkname` exists `status=0` is returned indicating success. Otherwise `status=-1` is returned.

```
DRMS_Record_t *record = drms_followlink(DRMS_Record_t *record, char *linkname, int *status)
```

Return a record containing the target record pointed to by the link named `linkname` in `record`. A dynamic link will be resolved to the record with the highest record number among those with primary index equal to the one in the link.

If no such target record exists or there is no link named `linkname` a NULL pointer is returned. If `status` is not NULL it is set to 0 for “success”, -1 for “no such target record”, and -2 for “no such link”.

```
DRMS_Record_t **record = drms_followlink_allversions(DRMS_Record_t *record, char *linkname,  
int *num_versions, int *status)
```

Works like `drms_followlink`, except that if the link is a dynamic link an array is returned of all record with primary index equal to the one in the link. The number of matching records is returned in `*num_records`.

5.1.5 Data Segments

```
DRMS_DataSegment_t drms_open_datasegment(DRMS_Record_t *record, char *segment_name, int  
mode)
```

blah

```
DRMS_DataSegment_t drms_close_datasegment(DRMS_Record_t *record, char *segment_name,  
int mode)
```

blah

5.2 Language bindings

5.2.1 C bindings

5.2.2 Fortran bindings

5.2.3 IDL bindings

5.2.4 Matlab bindings

A JSOC Internal functions

Currently a prototype is being developed, which contains the functions below that implement the internals of the JSOC library. Notice that some of the listed functions are fairly low level and will never have to be called by somebody implementing, e.g., a science pipeline module.

A.1 JSOC API types

Below is the contents of the header file `jsoc_types.h`, which defines the data structures for series, record, keywords, links, and data segments.

```
#ifndef _JSOC_TYPES_H
#define _JSOC_TYPES_H

#include "db.h"
#include "hash_table.h"

/* Constants */
#define JSOC_MAXNAMELEN      (255)
#define JSOC_MAXHASHKEYLEN  (JSOC_MAXNAMELEN+20)
#define JSOC_MAXUNITLEN     (20)
#define JSOC_MAXCLASSES     (8192)
#define JSOC_MAXQUERYLEN    (8192)
#define JSOC_MAXPATHLEN     (8192)
#define JSOC_MAXFORMATLEN   (20)
#define JSOC_MAXRANK        (255)
#define JSOC_MAXDATASEGMENTS (255)
#define JSOC_LAZY_INIT      (1L)
#define JSOC_MAXCOMMENTLEN  (2048)

/***** JSOC related types *****/

/* ("Keyword") values of keywords belong to one of the following
   simple classes. */
typedef enum {JSOC_TYPE_CHAR, JSOC_TYPE_SHORT, JSOC_TYPE_LONG,
             JSOC_TYPE_LOGLONG, JSOC_TYPE_FLOAT, JSOC_TYPE_DOUBLE,
             JSOC_TYPE_DATETIME, JSOC_TYPE_TIMESTAMP,
             JSOC_TYPE_STRING} JSOC_Simple_t;

#ifndef JSOC_TYPES_C
extern char *mdc_type_names[];
#endif

#define JSOC_MAXTYPENAMELEN (9)
#define JSOC_DATETIMELEN (32)
#define JSOC_TIMESTAMPLEN (32)

typedef union JSOC_Simple_Value
{
    char char_val;
    short short_val;
    long long_val;
    long long longlong_val;
    float float_val;
    double double_val;
    char datetime_val[JSOC_DATETIMELEN];
    char timestamp_val[JSOC_TIMESTAMPLEN];
    char *string_val;
} JSOC_Simple_Value_t;
```

```

typedef struct JSOC_Keyword_struct
{
    char name[JSOC_MAXNAMELEN];    /* Keyword name. */

    /* If this is an inherited keyword, islink is non-zero,
       and linkname holds the name of the link which points
       to the dataset holding the actual keyword value. */

    int islink;
    char linkname[JSOC_MAXNAMELEN]; /* Link name. */

    int column;                    /* Which column in the dataset table
                                   holds this keyword? */
    JSOC_Simple_t type;            /* Keyword type. */
    char format[JSOC_MAXFORMATLEN]; /* Format string for formatted input
                                   and output. */
    char unit[JSOC_MAXUNITLEN];    /* Physical unit. */
    int size;                      /* Size of keyword data in bytes.*/
    JSOC_Simple_Value_t value;     /* Keyword data. If the Keyword is used as part
                                   of a series template then value contains the
                                   default value. */

    char comment[JSOC_MAXCOMMENTLEN];

} JSOC_Keyword_t;

/* Links to other objects from which keyword values can be inherited.
   A link often indicates that the present object was computed using the
   data in the object pointed to.
*/

typedef enum { SIMPLE_LINK, QUERY_LINK } JSOC_Link_type_t;
typedef struct JSOC_Link_struct
{
    char name[JSOC_MAXNAMELEN];    /* Link name. */
    char target_series[JSOC_MAXNAMELEN]; /* Series pointed to. */
    int id, version;
    int column;                    /* Column in main series table where the
                                   target id is found; version is found
                                   in (column+1). */

    JSOC_Link_type_t type;
    char where_clause[JSOC_MAXQUERYLEN];
    char comment[JSOC_MAXCOMMENTLEN];
} JSOC_Link_t;

/*
   The data descriptors hold basic information about the in-memory
   representation of the data.
*/
typedef enum { JSOC_BINARY, JSOC_FITZ, JSOC_FITS } JSOC_Storage_Protocol_t;
typedef struct JSOC_Data_Locator_struct
{
    long long dsindex;    /* Index to the storage management table in
                           the database. */
    char *path;          /* If non-null points to the resolved path. */

```

```

    JSOC_Storage_Protocol_t storage_protocol;
} JSOC_Data_Locator_t;

typedef struct JSOC_Data_struct
{
    JSOC_Simple_t type;          /* Type of the observable. */
    char unit[JSOC_MAXUNITLEN]; /* Physical unit. */
    int size;                    /* Size in bytes of the elements. */
    int naxis;                   /* Number of dimensions. */
    int axis[JSOC_MAXRANK];      /* Size of each dimension. */
} JSOC_Data_t;

/* Data descriptor for a dataset. The data part of a dataset consists
   of a collection of observables. Each observable is an n-dimensional
   array of a simple type. */
typedef struct JSOC_Data_struct
{
    JSOC_Data_Locator_t location; /* Data set location and storage protocol. */
    int num_obs;                  /* Number of observables. */
    JSOC_Data_t obs[JSOC_MAXDATASEGMENTS]; /* An array of descriptors.
                                           Each layer represents an
                                           n-dimensional array of a
                                           single observable.*/
} JSOC_Data_t;

/* An in-memory slice of an observable. If the full observable is
   the n-dimensional array
       A(0:(obs->axis[0]-1),0:(obs->axis[1]-1),...,0:(obs->axis[obs->naxis-1]-1)
   then the memory location pointed to by data holds the slice
       A(start[0]:end[0],start[1]:end[1],...,start[obs->naxis-1]:end[obs->naxis-1])
   stored in column major order.
*/
typedef struct JSOC_Data_Slice_struct
{
    JSOC_Data_t *obs;
    int start[JSOC_MAXRANK];
    int end[JSOC_MAXRANK];
    void *data;
} JSOC_Data_Slice_t;

/* Datastructure holding a single data record. */
typedef struct JSOC_DataRecord_struct
{
    struct JSOC_Env_struct *env; /* Pointer to global JSOC environment. */

    /* The following three fields uniquely identify the object: */
    char seriesname[JSOC_MAXNAMELEN]; /* Name of series this dataset belongs to. */
    int id, version; /* (id, version) is a unique identifier
                     of this object within its series. */
    char hashkey[JSOC_MAXHASHKEYLEN]; /* Hash key is a string containing
                                       "seriesname_<id>_<version>". */

    /* Dirty flags */
    int keyw_dirty; /* Have the keywords been modified? */
    int link_dirty; /* Have the links been modified? */
    int data_dirty; /* Have the data arrays been modified? */

```

```

/* Keywords. */
int num_keyw, max_keyw;      /* Number of keywords. */
JSOC_Keyword_t *keyw;        /* Array of keywords. */
Hash_Table_t keyw_hash;      /* Hash table for fast lookup of keywords. */

/* Links. */
int num_link, max_link;      /* Number of links. */
JSOC_Link_t *link;           /* Array of links. */
Hash_Table_t link_hash;      /* Hash table for fast lookup of links. */

/* "Image" data. */
JSOC_Data_t *data;           /* Data descriptor. */
} JSOC_DataRecord_t;

typedef struct JSOC_Series_struct
{
    int tapegroup;
    int chunksize;
    int archive;
    char author[JSOC_MAXCOMMENTLEN];
    char owners[JSOC_MAXCOMMENTLEN];
    char description[JSOC_MAXCOMMENTLEN];
    JSOC_DataRecord_t template;
} JSOC_Series_t;

typedef struct JSOC_Env_struct
{
    int errno;                 /* Error flag. */
    DB_Handle_t *db;           /* Database connection handle. */

    /* Series cache data structures. */
    int num_series;            /* Total number of series listed in the
                                database. */
    char *init_tag;            /* Array of tags indicating which
                                series templates have been populated
                                with data from the database. */
    JSOC_DataRecord_t *series_cache; /* Cache array of series templates for
                                all series in the system. */
    Hash_Table_t series_hash;    /* Hash table for mapping series names
                                to indices in the series_cache array. */
    DB_Text_Result_t *series_names; /* Table of all seriesnames returned from
                                the database. These strings are used as
                                keys in the hash table. */

    /* DataRecord cache data structures. */
    int num_ds;                /* Number of datasets in memory. */
    int max_ds;                /* Max number of datasets in cache
                                memory. */
    char *ds_freelist;          /* List of free slots in the
                                ds_cache. */
    int ds_firstfree;           /* Index of first free slot in the
                                ds_cache */
    JSOC_DataRecord_t *ds_cache; /* Array of all datasets currently
                                in memory. */
    Hash_Table_t ds_hash;       /* Hash table for mapping dataset
                                identifier (seriesname, id, version) to
                                indices in the ds_cache array. */
} JSOC_Env_t;

```

```

/* Return enum value for a simple type given its name. */
JSOC_Simple_t mdc_str2type(char *);
const char *mdc_type2str(JSOC_Simple_t type);
int mdc_copy_db2mdc(JSOC_Simple_t mdc_type, JSOC_Simple_Value_t *mdc_dst,
                   DB_Type_t db_type, char *db_src);
DB_Type_t mdc2dbtype(JSOC_Simple_t type);
int mdc_sizeof(JSOC_Simple_t type);

```

A.2 JSOC environment functions

```

/* - Open authenticated data base connection.
   - Retrieve master series lists.
   - Build hash table over seriesnames. The series templates
     will be built on demand by querying the master keyword, link
     and observable tables.
   - Initialize datarecord cache and hash table. */
JSOC_Env_t *jsoc_initialize(const char *host, const char *user,
                           const char *password, const char *dbname);

/* - If commit_dirty==1 then commit all modified datarecords in the cache
   to the database.
   - Close database connection and free JSOC data structures. */
int jsoc_shutdown(JSOC_Env_t *env, int commit_dirty);

/* Commit all modified datarecords in the cache to database. */
int jsoc_commit_dirty(JSOC_Env_t *env);

/***** Datarecord cache operations. *****/

/* Return a cache slot to the free list and update firstfree. */
void jsoc_env_release_drcache_slot(JSOC_Env_t *env, int index);

/* Get the index of the first free slot in the
   dr cache and mark it used. If the cache is full
   double its size. */
int jsoc_env_get_drcache_slot(JSOC_Env_t *env);

/* Add or remove datarecords from the JSOC environment. */
int jsoc_env_remove_dr(JSOC_Env_t *env, JSOC_Datarecord_t *dr);

```

A.3 JSOC series functions

```

/* Parse a JSOC Series Definition string to a JSOC_Series
   object (a record template plus global series info). */
JSOC_Series_t *jsoc_parse_description(JSOC_Env_t *env, char *desc)

/* Given a JSOC series object execute the SQL code required to
   generate the tables and global table entries in the data base
   to represent the series. */
int jsoc_create_series(JSOC_Env_t *env, JSOC_Series_t *series)

/* Given a JSOC series object corresponding to an existing series
   execute the SQL code required to update the database tables
   and global entries to reflect any changes to the series
   definition. */
int jsoc_update_series(JSOC_Env_t *env, JSOC_Series_t *series)

```

A.4 JSOC record functions

```
/* Retrieve a data record from the database
JSOC_DataRecord_t *jsoc_dr_retrieve(JSOC_Env_t *env, const char *seriesname,
                                     int id, int version);

/* Commit a modified data record to the database. */
int jsoc_dr_commit(JSOC_Env_t *env, JSOC_DataRecord_t *dr);

/* Retrieve a linked datarecord. */
JSOC_DataRecord_t *jsoc_dr_follow_link(JSOC_Env_t *env, JSOC_DataRecord_t *dr,
                                       const char *linkname);

/* Query to find data records from a series satisfying a given condition. */
JSOC_DataRecord_t *jsoc_dr_query(JSOC_Env_t *env, const char *seriesname,
                                 const char *condition);

/* Assign the datarecord the next unique sequence number from
the database. The version number is set to 1. */
int jsoc_dr_assign_next_id(JSOC_Env_t *env, JSOC_DataRecord_t *dr);

/* Return a pointer to a copy of the series template whose
id has been set to the next unique sequence number in the database. */
JSOC_DataRecord_t *jsoc_dr_new(JSOC_Env_t *env, const char *seriesname);

/* Return a pointer to the series template. This template is used for
building new data records belonging to this series. */
JSOC_DataRecord_t *jsoc_dr_template(JSOC_Env_t *env, const char *seriesname);

/* Return a pointer to a copy of the series template. cache_index is set
to the number of the slot in the JSOC datarecord cache used. */
JSOC_DataRecord_t *jsoc_dr_allocate(JSOC_Env_t *env, const char *seriesname,
                                   int *cache_index);

/* Free a data record structure. */
int jsoc_dr_free(JSOC_Env_t *env, JSOC_DataRecord_t *dr);

/* Deep copy a data record. */
int jsoc_dr_copy(JSOC_Env_t *env, JSOC_DataRecord_t *dst, JSOC_DataRecord_t *src);
```

A.5 JSOC keyword functions

```
/* Versions with type conversion. */
int jsoc_keyw_get_char(JSOC_DataRecord_t *dr, const char *keyw_name, int *value);
int jsoc_keyw_get_int(JSOC_DataRecord_t *dr, const char *keyw_name, char *value);
int jsoc_keyw_get_float(JSOC_DataRecord_t *dr, const char *keyw_name, float *value);
int jsoc_keyw_get_double(JSOC_DataRecord_t *dr, const char *keyw_name, double *value);
int jsoc_keyw_get_string(JSOC_DataRecord_t *dr, const char *keyw_name, char **value);
/* Generic version. */
int jsoc_keyw_get(JSOC_DataRecord_t *dr, const char keyw_name, JSOC_Simple_Value_t *value);

/* Versions with type conversion. */
int jsoc_keyw_set_char(JSOC_DataRecord_t *dr, const char *keyw_name, char value);
int jsoc_keyw_set_int(JSOC_DataRecord_t *dr, const char *keyw_name, int value);
```



```

int jsoc_keyw_set_float(JSOC_DataRecord_t *dr, const char *keyw_name, float value);
int jsoc_keyw_set_double(JSOC_DataRecord_t *dr, const char *keyw_name, double value);
int jsoc_keyw_set_string(JSOC_DataRecord_t *dr, const char *keyw_name, char **value);

/* Use keyword hash table to quickly locate the data structure
   for specific keyword. */
JSOC_Keyword_t *jsoc_keyword_lookup(JSOC_DataRecord_t *dr, const char *keyw_name);

```

A.6 JSOC link functions

```

/* Define a simple query by specifying the data record pointed to. */
int jsoc_link_set_static(JSOC_Env_t *env, JSOC_DataRecord_t *dr, const char *link_name,
                        const char *target_series, int id, int version);

/* Define a query-link by specifying the target series and query. */
int jsoc_link_set_query(JSOC_Env_t *env, JSOC_DataRecord_t *dr, const char *link_name,
                       const char *target_series, query);

/* Bind a query-link to a specific datarecord by evaluating the link query
   on the target series table. Notice that this function is just a no-op
   when applied to a static link. */
int jsoc_link_resolve(JSOC_Env_t *env, JSOC_Link_t *link);

/* Return list of all datarecordr linked to *dr. */
JSOC_DataRecord_t *jsoc_dr_get_all_links(JSOC_DataRecord_t *dr);

```

A.7 JSOC data segment functions

[MISSING]

B JSOC Database Interface Layer (DIL)

This section describes a database interface layer (DIL) used to implement the JSOC. The DIL was created to provide a unified call level C interface to multiple database backends. So far, DIL contains support for Oracle 10g (via the Oracle OCI interface), and the two open source databases MySQL (version 4.1) and PostgreSQL (version 7.4.x).

B.1 DIL initialization and connect functions

```

/* Establish authenticated connection to database server. */
DB_Handle_t *db_connect(const char *host, const char *user,
                       const char *passwd, const char *db_name);

/* Disconnect from database server. */
void db_disconnect(DB_Handle_t *db);

```

B.2 DIL data manipulation functions

```

/* SQL data manipulation statement with fixed input and no output. */
int db_dms(DB_Handle_t *db, int *row_count, const char *query_string);

/* SQL data manipulation statement with variable array input and no output. */
int db_dmsv(DB_Handle_t *dbin, int *row_count, const char *query_string,
            int n_rows, ...);

```

B.3 DIL column types and query result types

```
/* Generic column types. */
typedef enum DB_Type_enum
{
    DB_CHAR, DB_INT1, DB_INT2, DB_INT4, DB_INT8,
    DB_FLOAT, DB_DOUBLE,
    DB_STRING, DB_VARCHAR
} DB_Type_t;

/* Binary query result column. */
typedef struct DB_Column_struct
{
    char *column_name;    /* Name of the column. */
    DB_Type_t type;       /* The data type. */
    unsigned int num_rows; /* Number of rows in the column. */
    unsigned int size;     /* Size of data type. */
    char *data;           /* Array of type "type" holding the column data.
                           The total length of *column_data is num_rows*size.
                           */
    signed short *is_null; /* An array of flags indicating if the field
                           contained a NULL value. */
} DB_Column_t;

/* Binary query result table. */
typedef struct DB_Binary_Result_struct
{
    unsigned int num_rows; /* Number of rows in result. */
    unsigned int num_cols; /* Number of columns in result. */
    DB_Column_t *column;
} DB_Binary_Result_t;

/* Text query result table. */
typedef struct DB_Text_Result_struct
{
    unsigned int num_rows;
    unsigned int num_cols;
    char **column_name; /* Name of the column. */
    int *column_width; /* Max width of the column. */
    char *buffer; /* buffers holding the results. On buffer per row. */
    char ***field; /* field[i][j] is a string contained in the i'th row
                   and j'th column of the result. */
} DB_Text_Result_t;
```

B.4 DIL query functions

```
/* SQL query statement with result returned as table of strings. */
DB_Text_Result_t *db_query_txt(DB_Handle_t *db, const char *query_string);

/* SQL query statement with result returned as table of binary data. */
DB_Binary_Result_t *db_query_bin(DB_Handle_t *db, const char *query_string);
```

B.5 DIL query result functions

```
/* Functions for extraction the field values from a binary result table. */
char *db_binary_field_get(DB_Binary_Result_t *res, unsigned int row,
                          unsigned int col);
int db_binary_field_is_null(DB_Binary_Result_t *res, unsigned int row,
                           unsigned int col);
DB_Type_t db_binary_column_type(DB_Binary_Result_t *res, unsigned int col);
void db_print_binary_field_type(DB_Type_t dbtype);

/* with conversion... */
char db_binary_field_getchar(DB_Binary_Result_t *res, unsigned int row,
                             unsigned int col);
int db_binary_field_getint(DB_Binary_Result_t *res, unsigned int row,
                           unsigned int col);
float db_binary_field_getfloat(DB_Binary_Result_t *res, unsigned int row,
                               unsigned int col);
double db_binary_field_getdouble(DB_Binary_Result_t *res, unsigned int row,
                                 unsigned int col);
void db_binary_field_getstr(DB_Binary_Result_t *res, unsigned int row,
                           unsigned int col, int len, char *str);

/* Formated printing of table of results. */
void db_print_binary_result(DB_Binary_Result_t *res);
void db_print_binary_field(DB_Type_t dbtype, int width, char *data);
int db_sprint_binary_field(DB_Type_t dbtype, int width, char *data, char *dst);
int db_binary_default_width(DB_Type_t dbtype);
void db_print_text_result(DB_Text_Result_t *res);

/* Free result buffers allocated by db_query functions. */
void db_free_binary_result(DB_Binary_Result_t *db_result);
void db_free_text_result(DB_Text_Result_t *db_result);
```

B.6 DIL transaction functions

```
int db_commit(DB_Handle_t *db);
int db_start_transaction(DB_Handle_t *db);
int db_rollback(DB_Handle_t *db);
```

B.7 DIL sequence functions

```
unsigned long db_sequence_getnext(DB_Handle_t *db, const char *tablename);
unsigned long db_sequence_getcurrent(DB_Handle_t *db, const char *tablename);
```

B.8 DIL type conversion functions

```
char dbtype2char(DB_Type_t dbtype, char *data);
short dbtype2short(DB_Type_t dbtype, char *data);
long dbtype2long(DB_Type_t dbtype, char *data);
long long dbtype2longlong(DB_Type_t dbtype, char *data);
float dbtype2float(DB_Type_t dbtype, char *data);
double dbtype2double(DB_Type_t dbtype, char *data);
void dbtype2str(DB_Type_t dbtype, char *data, int len, char *str);
```

B.9 Example program

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <time.h>
#include "db.h"

#define SERVER "hmi0"
#define USER "rasmus"
#define PASSWD "sql4rml"
#define DBNAME "rtest"

/* Open a database connection, create a table, insert records into it,
   drop the table and close the connection. */
int main(int argc, char *argv[])
{
    int i;
    DB_Handle_t *db;
    DB_Binary_Result_t *result;
    DB_Text_Result_t *tresult;
    char query[8192];
    int row_count, nrows;
    int *a;
    float *b;
    char *c;

    /* Ask the user how many records to create. */
    printf("Number of records: ");
    scanf("%d",&nrows);

    a = safemalloc(nrows*sizeof(int));
    b = safemalloc(nrows*sizeof(float));
    c = safemalloc(20*nrows*sizeof(char));
    /* Authenticate and connect to the database. */
    if ((db = db_connect(SERVER,USER,PASSWD,DBNAME))==NULL)
    {
        fprintf(stderr,"Couldn't connect to database.\n");
        return 1;
    }

    /* Drop the table 'test'. */
    db_dms(db, &row_count, "drop table test");

    /* Create a new table called 'test' */
    if (db_dms(db, &row_count, "create table test (a integer, "
        " b float, c varchar(20))"))
    {
        fprintf(stderr,"Couldn't create table.\n");
        goto bailout;
    }
}
```

```

/* Start a new transaction. */
db_start_transaction(db);

/* Insert a single row. */
if (db_dms(db, &row_count,
    "insert into test values (-2, NULL, 'blah vlah')"))
{
    fprintf(stderr,"Couldn't insert into table.\n");
    goto bailout;
}

/* Insert many rows into test. */
for (i=0; i<nrows; i++)
{
    a[i] = i;
    b[i] = 3.1415f+(float)i;
    sprintf(&c[i*20], "%-19s","Hello World!");
}
if (db_dmsv(db, &row_count, "insert into test values (?, ?, ?)",nrows,
    a,sizeof(int), DB_INT4,
    b, sizeof(float), DB_FLOAT,
    c,20,DB_STRING))
{
    fprintf(stderr,"Insert failed.\n");
    goto bailout;
}
/* Commit transaction. */
db_commit(db);

/***** Test various queries. *****/

/* Test summation. */
sprintf(query,"select sum(to_float(a)),sum(b) from test where a>%d and a<=%d",
    nrows/2,3*nrows/5);
printf("\nPerforming query: %s.\n",query);
tresult = db_query_txt(db, query);
printf("Result has %u rows and %u columns.\n",tresult->num_rows,
    tresult->num_cols);
db_print_text_result(tresult);
db_free_text_result(tresult);

/* Test summation again with binary return value. */
printf("\nPerforming query: '%s'\n", query);
result = db_query_bin(db, query);
printf("Result has %u rows and %u columns.\n",result->num_rows,
    result->num_cols);
db_print_binary_result(result);
db_free_binary_result(result);

/* Try a query that returns all columns for rows with a<40. */
sprintf(query,"select * from test where a<40");
printf("\nPerforming query2: '%s'\n",query);
result = db_query_bin(db, query);
printf("Result has %u rows and %u columns.\n",result->num_rows,

```

```

        result->num_cols);
db_print_binary_result(result);
db_free_binary_result(result);

/* Disconnect from the database. */
db_disconnect(db);
return 0;
bailout:
db_disconnect(db);
return 1;
}

```

C Acronyms

Acronym	Meaning
API	Application Programming Interface
JSOC	Joint Science Operations Center
JSD	JSOC Series Definition
DRMS	Data Record Management System
SUMS	Storage Unit Management System
DIL	Database Interface Layer