Use of DataSet Names

Philip H. Scherrer

SOI-TN-112 - 31 March 1993

Revised Sept 1995

Contents

Formal syntax of Dataset Names

Formal Syntax of Dataset Names

Within the SOI SSSC environment we have adopted a data naming standard which is used to identify "Datasets". While the term *dataset* has been used in many contexts, we hereby appropriate the term **Dataset** for our own purposes and will attempt to use it in the capitalized form whenever we are referring to SOI Datasets.

Notation for formal syntax of Dataset names:

Name parts are shown in italics.
Bold characters (letters or punctuation) not in italics are literal.
The symbol "::=" should be read as "consists of".
Brackets { } denote optional parts.
The symbol "<>" means concatenation.
A "**|**" should be read as "or".

Dataset names are built of normal names and numbers. Dataset names are case sensitive.

name ::= *alphabetic* **|** *name*<>*alphanumeric*
number ::= *digit* **|** *number*<>*digit*
number_range ::= *number*{**-***number*}

In particular, a Dataset name is built of six elements:

prog_name ::= *name*
level_name ::= *name*
series_name ::= *name*
prog_number ::= *number*

level_number ::= *number*
series_number ::= *number*

Prog_numbers and level_numbers are restricted to single numbers, but in some uses series_numbers may be combined to specify a set of series_numbers. In general a series_number list may be a comma delimited set of number ranges.

series_number_range ::= *number_range*
series_number_list ::= *series_number_range*{**,***series_number_range*}

The basic elements are combined with class keywords derived from the more general data naming concepts developed earlier in the SOI program. At present only these three data naming classes are allowed for SSSC Datasets.

program_selector ::= **prog:***prog_name{***[***prog_number***]***}*
level_selector ::= **level:***level_name{***[***level_number***]***}*
series_selector ::= **series:***series_name{***[***series_number***]***}*

Finally, the basic elements are combined to specify Datasets. A Dataset is a member of a DataSeries. A DataSeries is often taken to be an ordered set of Datasets where each Dataset contains data for, say, an interval of time such as an hour while the DataSeries contains all of the data for all of the hours available. Program input specifications allow a selection of Datasets from a particular DataSeries to be specified in a compact manner described here as a DataSeries_range. Finally, a DataCollection is just a list of DataSeries_ranges. A DataCollection is allowed as input (or output) from any SSSC pipeline analysis module that uses the standard ARGLIST functions to communicate with the runtime environment.

DataSeries_name ::= *program_selector*{**,***level_selector*}**,series:***series_name*
Dataset_name ::= *DataSeries_name{***[***series_number***]***}*
DataSeries_range ::= *DataSeries_name*{**[***series_number_list***]**}
DataCollection_name ::= *DataSeries_range*{**;***DataSeries_range*}

A Dataset is the atomic unit of data managed by the SOI SSSC DSDS system. Thus any program run in the SSSC pipeline environment can only obtain and save data in units of Datasets.

In addition to specifying Datasets, the naming syntax allows a specification of a selection of data within a particular Dataset

var_list ::= *name*{**,***var_list*}
record_range ::= *range*
axis_range ::= *range*
array_slice ::= *axis_range*{**,***array_slice*}
slice_selector ::= *record_range*{**,***array_slice*}
data_selector ::= **sel:**{*var_list*}{**[***slice_selector***]**}

Notes.

If a data specification is a SSSC Dataset name or DataCollection it will always begin with "prog:". The basic unit of data that is handled by the **SOI DSDS** is a **Dataset.** While a **DataCollection** may then be specified as a set of **DSDS Datasets**, the chunk that is processed at a time is a single Dataset.

While in the earlier data naming description (SOI TN xxx) any of the number sections could have a range of values specified, only the series_number may be expressed as a list for SSSC Datasets. If multiple prog_numbers or multiple level_numbers are desired as part of a DataCollection specification they must be explicitly enumerated. This is not viewed as a restriction since we have adopted a particular meaning for the level_number field which does not naturally require a general list syntax.

1 USAGE

The formal name specification for SOI data product names does not give sufficient understanding of the intended usage to allow either real use or development of the system. Therefore, the following description is intended to show the intent of the way to use this name format.

The purpose of the SOI naming format is to enable identification of chunks of SOI and related data independent of the particular storage location (disk, tape, etc.) or the particular storage protocol (FITS, CDF, etc.).

Certain rules of usage and limitations are designed to make the actual use of SOI data as practical as possible while bounding the problem of the DSDS and the pipeline and exploratory analysis programs in handling the expected very large data collection.

One of the goals is to enable the development of program modules that can be used in a production pipeline environment, in stand-alone analysis programs written to the same standards as the pipeline programs, and in GUI analysis environments such as SGI Explorer, Khoros, IDL, or a TAE driven analysis front-end. Since these analysis environments provide their own view of the world and each have differing expectations of the ability of computing modules to gain knowledge of their surroundings, we have decided to build SOI SSC analysis programs in two parts or levels. Actually, we have divided one of these levels into further divisions to (hopefully) enhance program maintenance capabilities but that division is hidden for the purposes of this discussion.

We call these two levels the “interface level” and the “strategy level.” The interface level is considered to be a “wrapper” around the strategy module that handles ALL connections with the program's environment. The goal is to produce a single interface level linkable module that is appropriate to make a unix "exec"able program suitable for each analysis environment needed. The particular environment connections that we will shield from the strategy modules include the DSDS catalog database, staging SOI datasets by the DSDS, command line parameters, stderr error logging, stdout message printing, program return values, etc.

This isolation is accomplished by defining a fixed calling interface for all strategy modules. This calling interface is similar to the main program calling interface in unix which provides connections with the command line and the programs environment variables. Since the various wrappers we will need will in general not want strategy modules to send output to the pre-opened i/o streams, but will want to do particular handling of communications with the user under their own control, we add some message i/o like functions in addition to handling “command line” control parameters.

Strategy modules are where the action is. It is the strategy modules that actually are written with specific knowledge of the particular problem at hand. The interface level has simply performed some services for the strategy modules that provide a clean connection with a virtual environment.

In particular, strategy modules communicate with the interface level in the following ways:

- The strategy module code includes pre-defined lists of expected control or command line arguments. These arguments are provided as initializations of an external scope array of structures. Each member of the array describes one expected control parameter. the strategy module specifies (at compile time) the key name that will be used for that parameter, the default value to use if the parameter is not specified (or a flag indicating that the parameter must be specified before starting the strategy level module), and probably we will also supply max and min appropriate values and the type of parameter so that a GUI can do some first level validity checking (or radio-dial building).
- The strategy module includes the definition of an external pointer named DOIT which is initialized to point to the strategy module itself. The interface module will use that external scope variable to invoke the strategy module.
- The interface module will provide a KEYlist which is a linked list of structures containing key_name:value pairs. It will examine the list of expected arguments, the command line, shell environment, possibly files in $HOME, and possibly files in other standard places.
- The interface module will provide an empty KEYlist for return values from the strategy module that will be reported to the calling environment.
- The interface module will provide a function which takes a string and puts it into a history log appropriate to the interface.
- The interface module will provide a function which takes a string and puts it into an error log appropriate to the interface. These string functions will take the same kinds of arguments as printf. In the shell callable interface module in cases where printf is a function rather than a macro, the history function will probably just be printf. These functions could actually be provided via external function pointers just like the strategy module is provided to the interface module. That might actually be a more extensible model.
- The interface module may later provide other “services.” To allow graceful upgrading of interface modules and strategy modules in a working system, the strategy module will also provide an external variable called STRAT_VERSION (or something like that) which the interface module can use to determine the expectations of the strategy module.

The expected flow of logic in the interface module would be something like:

1. Build empty KEY lists for control and return parameters.
2. Process the argument list provided by the strategy module.
   a. If simple numeric type, get it from the caller, maybe check it, and add it or its default to the KEYlist.
   b. If a DATASET type, the interface module must do a bit more.
      1. Call the dataset name parse routine which splits a dataset name into its component parts.
      2. Use the prog member_name and the environment to determine the directory naming rule for datasets in that prog.
      3. Put the dataset name and the directory template or rule into the KEYlist.
3. Call the strategy module (which returns a status) with the KEYlists and string reporting functions.
4. On return from the strategy module, check status and report failure to calling environment.
5. Examine the return KEYlist and report any values present to the calling environment.

2 TEMPLATE

A **dataseries** is a collection (one or more) files that is handled as a unit by the DSDS. The DSDS or a distribution of data from the DSDS or a dataseries to be supplied to the DSDS must reside within a unix filesystem directory. There is a well defined mapping between a **dataseries_name** and a directory. That mapping is via a rule specified in a template string. The template is a concatenation of literal characters and place-holders for class **member_names** and **member_numbers**. The rule for use of the template is as follows:

> Any literal characters are copied unchanged.
>
> Any construct **{class_name}** is replaced by the matching **member_name**.
>
> Any construct **{#class_name}** is replaced by the matching **member_number** converted to a string via a sprintf with format "%d".

Note that an empty template produces an empty string for the directory name. Thus, an empty template can be used to always refer to the current directory.

The **dataset_name** processing function will operate on a KEYlist with a **key_root** name also provided. The prototype might be:

```
int parse_name( KEY *params, char *key_root);
```

and it might be used as:

```
if ( !parse_name(call_keys, "in"))
    complain_to_someone();
```

The **key_root** name, e.g. "in" will be the keyword used to locate the **dataset_name** in the KEYlist. All related keyword names will be built from the **root_key** by appending a suffix of the form "_parsed_name_part", e.g. the template itself will be available via the keyword "in_rule" if the **root_key** is "in".

The parse_name function that uses the template will take a **dataset_name** and do the following:

0. Look for pre-existing parameters **root_key**_dataseries and **root_key**_selector.
   if they are found, use them below, else:
1. Separate the **dataseries_name** and the **data_selector** parts of the **dataset_name** and make keylist entries **root_key**_dataseries and **root_key**_selector.
2. Separate the **dataseries_name** into its component classes. The **prog** class must always be present when the **dataseries_name** is not empty.

2.5 Build keylist entries for each class of the form, e.g. "in_prog", "in_level", "in_series". If such entries were already present in KEYlist use the ones from KEYlist rather than the just learned ones from the **dataseries_name**.

3. If the **prog** class is not there, i.e. there is no **dataseries_name** present, the directory name is an empty string. This means a **dataset_name** which does not start with **prog:** will default to the current directory.
4. Step through the template using the template rules above to construct a directory name.
5. The directory is added to the KEYlist with keyword **root_key**_wd, e.g. "in_wd".
6. If any of the **member_numbers** were **ranges** the name being parsed is a **datacollection_name** rather than a **dataset_name**. The parse routine should always prepare keyword entries of the form "in_series_sn", "in_series_fsn", and "in_series_lsn" where the "fsn" and "lsn" mean "first sequence number" and "last sequence number" and will be the same if a range was not given but rather a single sequence number. If a range was found, the "sn" parameter, e.g. "in_series_sn" should be set to an invalid sequence number, e.g. -1 or perhaps minus number in range. If any of the **member_numbers** were ranges, the eventual parse function return value should indicate that the name referred to a **datacollection** rather than a **dataset** by perhaps returning the product of all the range lengths so that the calling program can examine the return value to see if an SDS_open will be possible without further qualification of the name parts.

The function will then parse the **data_selector** part of the name. While we may someday expand the syntax to allow a list of variable names, FOR NOW WE WILL ONLY ALLOW ONE VARIABLE NAME in the **data_selector**. The keywords **root_key**_select and **root_key**_slice could contain the two parts of the selector. But I suggest that we do a bit more right here since we will need it later. That would be to split the slice spec into its component **axis_ranges**. Until we decide how to know the axis names, we can use only the first axis_range and leave the rest combined. Since the first **axis_range** present will be interpreted as a record or image number, I suggest we extract the following keyword parameters from the **data_selector** (using "in as an example):

in_select -> variable name
in_fsn -> first record number
in_lsn -> last record number
in_sn -> (in_fsn == in_lsn ? in_fsn : -(in_lsn - in_fsn +1))
in_slice -> any remaining part of the slice_selector.

In addition to building the working directory, e.g. "in_wd" the parse function must build a basename that will be the leading part of the actual data filename. The basename will be constructed from the **series_selector** by the fixed template:

"{series}.{#series}"

The resulting string will be stored in KEYlist as **"root_key**_basename", e.g. "in_basename".

At this point the parse function is done and can return.

3 ACTUAL FILE NAMES

There are several aspects to the actual file names associated with a **datacollection_name** or a **dataset_name**. It is a **dataseries_name** that maps to a set of files. A **dataset_name** then defines a set of files since it is contained within a single dataseries. A **datacollection** is a collection of sets of files. They will have to be handled one at a time. Thus the DSDS, the pipe, or the strategy modules will have to expand a **datacollection_name** into its implied set of **dataseries_names** prior to actually doing anything with them. I assume that the above parse function can be used in all three areas to decide what to do.

Once a specific **dataseries** is specified, the collection of files is EXACTLY the set that is matched if e.g. "wd" and "basename" were shell variables, by the shell wildcard expansion of:

$wd/$basename.*

The actual files containing data will depend on the data storage protocol. There are several cases that we can specify now: In the following I will use the shell variable example method above to have specific examples rather than using the **root_key**_word method.

FITS Tables

A column in a fits table maps directly to a dataset as described above. I suggest that we ignore the desired case of a complex dataset that describes multiple columns or fields from a fits table at the start. The fits table file which contains the simple dataset has the name:

$basename.fits

FITS images

Fits image files will be stored ONLY in simple fits files with one image per file. The image files, one per record number, will be:

$basename.$select.$sn.fits

CDF datasets

Cdf datasets are stored as a set of files. I believe they have a basename that matches the "name of the cdf" dataset. with several extensions appended in the usual way. If this is indeed the case, then the names are:

$basename.cdf_extensions

Associated files

Since the unit handled by the DSDS is determined by wildcard expansions of the basename within the working directory, we can easily allow additional files to “go along” with the actual data files. These could be history files:

$basename.history

processing exception logs:

$basename.errors

text describing the data product generation and usage:

$basename.text.ps

etc.

It would be the job of SDS_open to determine the actual protocol present by examining the files actually present by using fstat(3) or by consulting the keyword "in_type" which may have been provided. (Which must have been provided in the case of output dataseries.)

4 Problems:

At least one limitation of the description above is in describing datasets with more than one variable. Since we actually want to be able to do this at least as a **datacollection** but probably as a **dataset** (similar to the WSO DS datasets and to the real power of CDF datasets). To deal with such datasets at the low level requires complexity that is probably already being built into the SDS internal data file access functions, IDS. We should keep the simplification that an SDS_getdata (whatever it is called) returns a pointer to a simple datacube of a single variable, but how that selection is made is not clear to me. We have the notation possible to list the variable selects as a simple comma separated list of **data_selectors** as defined here. That does, however, leave the mapping onto files something that must be understood. Actually, maybe the present mapping works fine!

If it is a FITS table or CDF it still uses only basename to get the file name and if it is FITS image files, the variable name is used. The one problem I see is that we should require that the record number (or sn) range be identical for all variables part of one complex dataset. This is the one place where having the record number serial number be part of the series name rather than part of the variable name made more sense. There are so many advantages in the way this scheme makes the DSDS job well defined, that I would like us to think about a way to solve this problem within the present framework.

5 DSDS NOTES

A topic that I also wanted to discuss a bit is how the DSDS is used. I assume that the DSDS functions split into two logically separable areas: using a physical description of a dataset to identify the **dataseries_names** that contain the desired data, and quite apart, to distribute (internal and external) and ingest (internal and external) and archive **dataseries**.

We must list the **dataseries** that we will query on to build a list of keys for such queries. We can decide that just as there is a template for each **prog** since the **prog** determines the set of classes, there is a well defined set of search keys available for each **prog**.

Some of the keys are obvious: class **member_names** and **member_numbers**., **var_names**, **axis names, axis lengths, axis first and delta physical values, axis units.**

Other keys will include, **dates, access rights**, etc. I expect that there will be thousands or tens-of-thousands of **dataseries** vs. millions of images.

When new data is put into the DSDS it could be by one of two paths, operator action such as loading the raw telemetry data in which case only the tape label info is needed as keys, and from strategy modules called by shell or pipeline parents. When creating new data, strategy modules should:

1. Make a KEYlist containing the new name parts, directory must be consistent with template for the **prog**.
2. Call a function that builds a **dataseries == dataset name**
3. Build the datafiles with SDS functions.
4. Return a set of catalog key values in the return KEYlist

Then the calling program (shell or pipeline) should build a request to the DSDS to take this data and put it somewhere using the catalog keys provided to help find its name in the future.

If we want to install data pre-existing in some format, we can simply make a simple strategy module program that extracts the key info and returns.

6 DISTRIBUTED DATA

When we distribute data we will send only DSDS owned **dataseries**. We would package the dataseries with its name and also provide, together or apart, the template that we used to map to files. We could provide install instructions to extract the datafiles from the distribution media/file format using one of the templates. I can even imagine our own SFDU format that has the **dataseries_name**, the **dataseries**, and a sample template, and perhaps a shar file to extract it.