SUBJECT: **Specifying Data Products in the Herschel Interactive Analysis System**

PREPARED BY: **S.Guest**

DOCUMENT No: **SPIRE-RAL-DOC-001964**

ISSUE: 0.2 **Date:** **14th June 2004**

Change Record

ISSUE	DATE	
0.1 *draft*	10th February 2004	First draft
0.1 *draft 2*	5th March 2004	Second draft
0.1	18th March 2004	First wider distribution
0.2	14th June 2004	Latest numeric; quantities

Table of Contents

1 INTRODUCTION

1.1 Acronyms

API	Application Programming Interface
HDU	Header and Data Unit (FITS)
FITS	Flexible Image Transport System
IA	Interactive Analysis
SPG	Standard Product Generation
TBD	To Be Defined

1.2 Conventions used in this document

Words that are written in **bold** indicate special keywords that can be used in a product specification. The `courier` font is used to indicate that the text should be taken literally, for example an actual class name or name of a metadata item.

1.3 Purpose

Firstly, the purpose of this document is *not* to specify the standard data products that will be produced by the Herschel mission. That responsibility lies elsewhere.

The purpose of this document is to specify *how* to specify a data product for use within the Herschel Interactive Analysis framework. This will allow interfaces between processing modules to be defined in a standard, coherent and common way.

This specification is deliberately independent of programming languages, file formats, and database management systems. Guidelines for implementing it within specific environments are given in appendices.

2 DEFINITIONS

FITS analogies are used as examples in these definitions for clarity. The definitions are however completely independent of FITS format, and other export formats are also possible.

2.1 Metadata

A metadata item corresponds to a FITS header keyword. Metadata items consist of:

- The **name** of the item as a **string** keyword
- The **value** of the item
- A **description** of the item as a **string**
- A **quantity** of the item i.e. its units. The Java implementation uses the nanoTITAN Quantity Library for this (see Appendix A).

A **metadata value** has one of the following forms:

- **string** (Unicode sequence)
- **boolean**
- **double** (64-bit floating point)
- **long** (64-bit signed integer)
- **date** (including the time to millisecond resolution)

2.2 Data

A **data** item corresponds to a FITS array or column of a table. It is important to be clear that a **data** item is an *array* – single data items should be treated as **metadata**. Data can have the following types:

- **boolean**
- **byte** (8-bit signed)
- **complex** (64-bit signed pair)
- **double** (64-bit signed)
- **float** (32-bit signed)
- **integer** (32-bit signed)
- **long** (64-bit signed)
- **short** (16-bit signed)
- **string** (sequence of Unicode characters)

It is also necessary to specify the dimensionality of data arrays. This is done by appending "**-nd**", where '**n**' is the number of dimensions, so for example, **float-2d**. The maximum size of '**n**' is three, except for the **string** type, where it is one[1]. An array can only contain data of a single type (for example, you can't mix **float** and **integer** in a single array).

2.3 Dataset

A **dataset** corresponds to a FITS extension (eg a binary table), ie a non-primary HDU, though more complex structures are possible than are supported by the FITS standard. It is used to define data structures at a higher level than arrays, such as tables, images or spectra.

[1] The reason for this restriction is that it is what the current implementation supports. There does not appear to be any obvious need for string arrays of more than one dimension.

A **dataset** has these elements associated with it:

- **metadata**, though no particular items are required
- A **description** as a **string**

The following types of **dataset** may be specified:

- An **array dataset** that contains a **data** item (remember that a **data** item *is* an array). This can be used for images, cubes, single spectra etc. This corresponds to a FITS primary HDU or image extension. This should be used in preference to a **data** item when relevant **metadata** exists.
- A **table dataset** that contains one of more **columns** of **data** items. Specifically, a **column** is comprised of:
  - a **data** array of any dimension (not restricted to 1-d arrays)
  - a **description**, as a **string** (optional)
  - a **quantity** (optional)

  The types of **data** in the columns can be mixed. All columns must have the same length i.e. number of rows. This corresponds to a FITS binary table extension.
- A **composite dataset** that is composed of one or more **datasets**, which in turn could also be **composite** themselves. Note that this allows the possibility of hierarchically structured data. There is no equivalent in the FITS standard of this, though a proposal does now exist for hierarchical associations of related FITS files.

2.4 Product

A **product** is the highest level of data structure, similar to a **composite dataset**, but also providing a number of extra features. Despite the FITS analogies, there is no requirement that **products** ever have a *persistent* format such as a file or as an object in the database. Persistent in this sense just means that it is somehow "saved", and has a lifetime longer than a single IA session. These persistent formats should be regarded only as an external form of the **product**. It is therefore valid for a **product** to be purely *transient*, i.e. exist only in computer memory and never be saved. This data structure is suitable for standard[2] or non-standard products[3], and for calibration products.

A **product** is comprised of the following components:

- A **type** by which it can easily be identified[4], as a **string**
- A **description**, as a **string**
- **metadata**
- Zero or more **datasets**, i.e. more than one or none at all are both valid
- A processing **history**

It corresponds to an entire FITS file. The **metadata** belonging to the **product** correspond to the keywords in the primary header.

The following **metadata** items are required to be present in a **product**:

- `creator` as a **string**
- `creationDate` as a **date**
- `instrument` as a **string**
- `modelName` as a **string**

[2] In the SPG sense, i.e. "official" products output from a pipeline
[3] In the general sense, not the specific **product** sense of this specification
[4] Not currently implemented in `herschel.ia.dataset`

- `startDate` as a **date**
- `endDate` as a **date**

Note that the Herschel IA FITS interface automatically translates these names to the equivalent FITS keywords.

The definition of the processing **history** is currently beyond the scope of this document. The creation and maintenance of the **history** is the responsibility of the processing modules[5] that create and update the **product**. It is accepted that it would be useful to list the elements that must be contained in such a **history**, however this has not yet been defined.

3 EXAMPLE PRODUCT SPECIFICATION

It should be possible to achieve a reasonable degree of standardisation by consistently using the keywords and by following the rules defined in the preceding section. Here is a suggestion of how a product specification could be laid out:

```
product (type="spectrum", description="output of flat-fielding"):
     metadata:
          long obsid
          long bbid
     table dataset (description="time series"):
          metadata:
               date creationDate
               string equinox
          double-1d time (description="On-board time, epoch 1958")
          integer-1d dpuCount (description="DPU clock")
          float-1d specfarray001 (description="detector",
               quantity=Volts)
          …
          float-1d specfarray072 (description="detector",
               quantity=Volts)
     array dataset (description="detector mask"):
          boolean-1d mask[72]
```

The example deliberately leaves out the required **metadata** for the **product** as they are implicitly always present i.e. there is no need to repeatedly specify something that is always the same. The processing **history** is similarly always present.

It is currently assumed that such guidelines are sufficient, and that it is not necessary to describe a specification grammar in more formal terms.

[5] Herschel IA refers to these either as a **task** (command-line type synchronous, i.e. process to completion in a single execution), or a **process** (stream processing).

Appendix A. Java implementation

There is an existing Java implementation in the `herschel.ia.dataset` and `herschel.ia.numeric` packages. The former deals with **product**, **dataset** and **metadata**; the latter with **data**. These packages are designed to be convenient to use in a Jython as well as a Java environment. When a **product** is created, its required **metadata** items are automatically inserted, although only `creationDate` has its value set. The mappings between the keywords used in this document and the Java classes and interfaces are given below.

product and **dataset**		**metadata**	
product	`Product`	**boolean**	`BooleanParameter`
dataset	`Dataset`	**double**	`DoubleParameter`
array dataset	`ArrayDataset`	**long**	`LongParameter`
table dataset	`TableDataset`	**string**	`StringParameter`
composite dataset	`CompositeDataset`	**date**	`DateParameter`
metadata	`MetaData`		

data	**1-d**	**2-d**	**3-d**
boolean	`Bool1d`	`Bool2d`	`Bool3d`
byte	`Byte1d`	`Byte2d`	`Byte3d`
complex	`Complex1d`	`Complex2d`	`Complex3d`
double	`Double1d`	`Double2d`	`Double3d`
float	`Float1d`	`Float2d`	`Float3d`
integer	`Int1d`	`Int2d`	`Int3d`
long	`Long1d`	`Long2d`	`Long3d`
short	`Short1d`	`Short2d`	`Short3d`
string	`String1d`		

A **quantity** is implemented using the nanoTITAN Quantity library. See http://nanotitan.com/software/Libraries/quantity/index.htm for full details.

A.1 Jython Example

Here is the example product from section 3, coded in Jython (for illustration only!):

```
# Example product definition
# SG, March 2004
from herschel.ia.dataset import *
from herschel.ia.numeric import *
from java.util import Date
from nT.quantity.ElectricPotentialDifference import VOLTS

# Start with the main product
p = Product ("output of flat-fielding")   # no type supported yet
p.meta["obsid"] = LongParameter (1234)
p.meta["bbid"]  = LongParameter (0x89ab)

# Now create a table. TO DO: add quantities to the columns.
t = TableDataset ("time series")
t.meta["date"] = DateParameter (Date())   # current date and time
t.meta["equinox"] = StringParameter ("1950")
```

```
t["time"] = Column (Double1d ([1,2,3]), None, =”On-board time, epoch
1958”)
t["dpuCount"] = Column (Int1d ([10,20,30]), None, “DPU clock”)

# Generate the column names for the detectors. This could also be done with
# a loop, but this way demonstrates a powerful list-processing feature of
# Jython. lambda defines an anonymous function and map applies (maps) it
# over all the elements of a list.
ndets = 72
dets = map (lambda x,y: "%s%2.2i" % (x,y+1), \
           ["specfarray"]*ndets, range(ndets))

# data are zero in this example
for detector in dets: t[detector] = Column (Float1d(3), VOLTS, “detector”)

# Now use an array dataset for the mask
a = ArrayDataset (Bool1d(ndets), description="detector mask")

# Finally, put the datasets in the product.
# Here the descriptions are used as keys.
p.set (t.description, t)
p.set (a.description, a)
```

A.2 Java Example

Here is another version of the previous example, this time in Java. Again, this is for illustration purposes and no attempt has been made to make it more object-oriented!

```
import herschel.ia.dataset.*;
import herschel.ia.numeric.*;
import java.util.Date;
import java.text.*;
import nT.quantity.ElectricPotentialDifference;

/**
  * Example product definition
  * @author SG, March 2004
  */
public class FlatField {
  public static Product createOutput() {

    // Start with the main product, no type supported yet
    Product p = new Product ("output of flat-fielding");
    MetaData meta = p.getMeta();
    meta.set ("obsid", new LongParameter (1234));
    meta.set ("bbid", new LongParameter (0x89ab));
    p.setMeta (meta);

    // Now create a table
    TableDataset t = new TableDataset ("time series");
    meta = new MetaData();
    meta.set ("date", new DateParameter (new Date()));
    meta.set ("equinox", new StringParameter ("1950"));
    t.setMeta (meta);
    t.addColumn ("time", new Column (new Double1d (new double[] {1,2,3}),
            null, ”On-board time, epoch 1958”));
    t.addColumn ("dpuCount", new Column (new Int1d (new int[]{10,20,30}),
            null, “DPU clock”));
```

```
    // Add a column for each detector
    final int NDETS = 72;
    NumberFormat format = new DecimalFormat ("00");

    for (int i = 0; i < NDETS; i++) {
        String colName = "specfarray"+format.format (i+1);
        // data are zero in this example
        t.addColumn (colName, new Column (new Float1d(3),
            ElectricPotentialDifference.VOLTS, “detector”));
    }

    // Now use an array dataset for the mask
    ArrayDataset a = new ArrayDataset (new Bool1d(NDETS));
    a.setDescription ("detector mask");

    // Finally, put the datasets in the product.
    // Here the descriptions are used as keys.
    p.set (t.getDescription(), t);
    p.set (a.getDescription(), a);
    return p;
  }

  // test program
  public static void main (String[] args) {
    Product p = createOutput();
    System.out.println (p);
    System.out.println (p.get ("time series"));
    System.out.println (p.get ("detector mask"));
  }
}
```

Appendix B. Storing Products In Versant Databases

This section is written with the Versant Database Management System in mind. It is quite possible that the same principles also apply elsewhere.

B.1 Don't make data elements too big:

The current implementations of the data classes represent the data internally as Java primitive arrays. Versant treats these arrays as single objects. This has some implications where large arrays are concerned:

- Slow apparent startup time as the entire array has to be transferred across the network before a single element can be accessed.
- It is not possible to access the array without storing its entirety in memory, which can lead to poor performance and out-of-memory errors.

Exactly what constitutes "big" is hard to define, and dependent on the hardware specification. More investigation would be useful, but as a rough rule of thumb, array sizes bigger than a quarter of the available memory should be avoided.

More generally, split multidimensional arrays of significant size up when it makes sense. For example, a 2-D array can be represented as a 1-D array of row (or column) objects.

B.2 Use the provided API

There is an API in the `herschel.ia.io.dbase` package that should be used to create persistent database products from IA products. It is possible that in future it might be able to automatically perform certain optimisations such as the memory management discussed above. This package provides quite a low-level API, but it is planned to add one or more higher-level APIs to perform these functions.