Parameter Handling System

Jorgo Bakker, Jon Brumfitt and Marco Beijersbergen

16 December 1999

Abstract

The parameter handling system allows tasks to retrieve parameters which are specified on the command line. The parameters are defined in a Parameter Specification File. The current system utilises:
- default values
- complex expressions for allowable values
- parameters that can have child-parameters, depending on its value
- implicit setting of controlling parameters, if a child is activated on the command line

1 Introduction

Tasks may be configured by a set of parameters, which are entered on the UNIX command line. Every task has a Parameter Specification File which defines the name and type of each parameter and possibly a default value, allowable range, etc. An Application Programmers Interface (API) allows parameters to be retrieved by tasks written in either C++ or Fortran-90.

Parameters may also be entered using the Graphical User Interface (GUI). This reads the parameter specification file and provides a dialog window into which the user can enter parameters. The GUI then launches the task by generating a command line. For further information, see the documentation of the 'gui' package.

2 Command line syntax

Parameters are specified on the command-line in either of the following formats:

- GNU long option style: - -<param>=<value>
- FTOOLS XPI parameter interface style: <param>=<value>

The command line may only specify parameters which are defined in the parameter file.

Examples:
foo --ival=42 --rval=1.23          # Simple parameters
foo --sval=hello                   # Simple string
foo --sval='hello world'          # String with space
foo --slist='"one two" three'     # List of two strings

Arguments which contain spaces must be quoted. This also applies if the string has a leading or trailing space which must be preserved. For example:

    foo --sval=' hello world'

The shell’s file-name expansion mechanism cannot be used directly to specify a list of filenames. Filename expansion may be implemented within the parameter package, at a later date. In the meanwhile, the following example shows a work-around solution:

    foo --slist="'echo *.fits *.out'"

The single-quote character may be used in an argument, provided that it is correctly quoted or escaped. The following examples are both valid:

    foo --sval="can’t wait"
    foo --sval='can’t \’t wait’

If the parameters does not contain allowable values, as defined by the parameter specification file, an error message is given and the task is not run. The user should then correct the command-line, using the shell’s history mechanism.

Command-line options with a single ‘-’ are used for certain generic functions and are not considered as task parameters. See the documentation of the taskmain package for further details.

3 Parameter types

The following parameter types are supported:

bool  type that can have two values: true or false
int   a member of the set of whole numbers
real  a member of the set of rational and irrational numbers
string sequence of characters
directory name of the location of a list of files on a storage device
file  name of any file
dataset name of a SAS data file
block  general type for a section of data within a dataset
table  a block in a dataset, interpreted as a table
array  a block in a dataset, interpreted as an array

column  a column of data within a table of a dataset

angle  type that allows different formats of angle specification

time  type that allows different formats of time specification

Each parameter may be specified to contain either a scalar (single) value or a list of values:

list<bool>
list<int>
list<real>
list<string>
list<directory>
list<file>
list<dataset>
list<block>
list<table>
list<array>
list<column>
list<angle>
list<time>

The use of different types allows the parameter interface to check that the value is syntactically correct and, where appropriate, within an allowable range. It also allows the GUI to provide appropriate widgets for entering the values.

For example, a file-name is simply a string, as far as the task is concerned. However, using a separate type allows the GUI to present the user with a file dialog, instead of a simple string-edit field.

3.1  bool

Boolean values may be represented by any of the following symbols:

0, n, no, f, false
1, y, yes, t, true

For example:

taskname --foo=n --bar=yes
3.2 int

Example:

    taskname --foo=42 --bar=-123

3.3 real

Real parameters may be specified in simple decimal values or using exponent notation. For example:

    taskname --foo=1 --bar=1.23 --x=-1.23e-5

3.4 string

String parameters are used to represent text strings. Strings containing spaces should be enclosed in quotes. For example:

    taskname --foo=hello --bar='hello world'

3.5 directory

Directory parameters are used to specify a directory.

The directory may be expressed as a relative or absolute path. For example:

    bar
    ../foo/bar
    /data/foo/bar

3.6 file

File name parameters are used to express the name of any type of file. To specify a SAS data file, the dataset type is used. The SAS GUI uses a file dialog so that the user can select a file.

A file name parameter may specify an absolute or relative path. For example:

    bar.dat
    ../foo/bar.dat
    /data/foo/bar.dat

3.7 dataset

Data-set parameters are used to specify SAS datasets (a.o. FITS files).
The parameter may specify an absolute or relative path. For example:

\begin{verbatim}
bar.FIT
./foo/bar.FIT
/data/foo/bar.FIT
\end{verbatim}

There are additional parameter types which allow components of a dataset, such as blocks, tables, arrays and columns to be specified (see below).

The SAS GUI provides a special browser for examining and selecting datasets and their components.

### 3.8 block, table and array

Table parameters and array parameters allow a block (generic name of a table or array) and its associated dataset to be specified by a single parameter, using a colon-separated pair of the form `set:table` or `set:array`.

Block parameters are used where the value may be either a table or an array. This is appropriate for tasks which can perform generic operations, such as deleting a block.

Examples:

\begin{verbatim}
foo.FIT:R1SPE1     # Table R1SPE1 in set foo.FIT
/data/bar.FIT:FILTER_U  # Array FILTER_U in array bar.FIT
\end{verbatim}

The SAS GUI allows a block, table or array to be selected in a single operation, using a dataset browser. This is simpler than entering separate parameters for the dataset and block-name.

### 3.9 column

Column parameters allow a column and its associated table and dataset to be specified as a single parameter, using a colon-separated triple, of the form `set:table:column`.

For example:

\begin{verbatim}
foo.FITS:R1SPE1:OCDNODE  # Column OCDNODE in table R1SPE1 in set foo.FITS
\end{verbatim}

The SAS GUI allows a column to be selected in a single operation, using a dataset browser. This is simpler than entering 3 separate parameters for the dataset, table and column.

### 3.10 angle

Angle parameters are intended for right ascensions and declinations, where conversion from hours/minutes/seconds or degrees/minutes/seconds may be needed. For other angles, such as small offset angles, a real parameter is more appropriate. Note that the maximum range of the angle type is \([-360,360]\) degrees.

The angle can be entered in one of the following formats:
\(+|-\)xxdxxmxx[.xxx]s  (degrees, minutes and seconds)

xxhxxmxx[.xxx]s  (hours, minutes and seconds)

[\(+|-\)xx[.xxx]]  (decimal degrees)

where 'x' is a single digit, 'd', 'm', 's' and 'h' are literal characters and square brackets denote optional items. The number of digits shown is not significant.

Note: Angle parameters are accessed by the task as a real value, in degrees.

3.11 time

Time parameters allow times to be entered in a variety of different formats:

\(\text{xxxx-xx-xxTxx:xx[.xxx]}\) (FITS date and time)

\(\text{j\phantom{.}dxxxx.xxx}\) (Julian day)

\(\text{m\phantom{.}jd\phantom{.}xxx.xxx}\) (modified Julian day)

\(\text{yyyy.xxx.xxx}\) (decimal year - Not yet implemented)

\(\text{[-]xxx[.xxx]}\) (seconds since XMM reference time)

\(\text{xx \phantom{.}xxx \phantom{.}xx:xx:xx:xx\xxxx}\) (calendar format)

where 'x' is a single digit, 'm', 'j', 'd' and 'T' are literal characters and square brackets denote optional items. The number of digits shown is not significant.

Examples:

\(1997-10-13T00:00:00.000\) (FITS date and time)

\(\text{j\phantom{.}d2450734.5}\) (Julian day)

\(\text{m\phantom{.}jd50734}\) (modified Julian day)

\(-6.912\times10^6\) (seconds since XMM reference time)

\(\text{Mon Oct 13 00:00:00 1997}\) (calendar format)

Note: Times are accessed by the task as real values which represents the number of seconds since the XMM mission reference time.

3.12 list

Special types are provided for lists of each of the simple parameter types. List items are separated by spaces. For example:

\(\text{taskname --group='xmm sas user developer'}\)

String lists may use double-quotes to allow items which contain spaces or empty strings. For example, the following list contains 5 items ('"ab cd"', '"ef"', '".."', '"..", "String list"):

\(\text{taskname --list='"ab cd" ef '' ,, "String list"'}\)
4 Optional and mandatory parameters

Each parameter has a default value, which is used if no parameter is specified on the command line. A parameter may be either mandatory or optional. The default value and mandatory attribute are specified in the parameter file.

A mandatory parameter must be explicitly given on the command-line, otherwise an error message will be printed to prompt the user to re-enter the command. If an optional parameter is not specified, the default is used.

Note: Mandatory parameters have a default value for use in the GUI. The GUI assigns values to all parameters and does not treat mandatory parameters specially.

5 Constraints on values

If you specify a parameter on the command-line, the value of the parameter must be of the appropriate type.

Optionally the parameter file may define constraints on accepted values of that type. For example, for integer types this may be all positive whole numbers.

6 Errors

This section documents warnings and errors generated by this task (if any). Note that warnings and errors can also be generated in the SAS infrastructure libraries, in which case they would not be documented here. Refer to the index of all errors and warnings available in the HTML version of the SAS documentation.

ParamCorrupted (error)
Parameter space got corrupted. Accessing any parameter may stop your program working.

ParamFileOpen (error)
Could not open the parameter file for reading. Either the parameter does not exist or it is not readable.

ParamFileInvalid (error)
The contents of the parameter file is invalid.

ParamFileNotXml (error)
The contents of the parameter file violates the XML syntax

ParamIndexRange (error)
The list parameter is accessed to return an item at a position, that is out of range

ParamInvalidConstraints (error)
The syntax or the contents of the constraints expression is wrong

ParamMandatory (error)
Lists all parameters that still need to be specified on the command-line
**ParamNotFound** *(error)*
Requested a parameter that is not found in the parameter space of this task

**ParamOutOfRange** *(error)*
The value of this parameter is outside the range as expressed in the constraints

**ParamNotList** *(error)*
The parameter is incorrectly accessed as a list

**ParamWrongQuotes** *(error)*
Mismatch of quotes in the value of the parameter

**ParamWrongType** *(error)*
The value of this parameter has the wrong type

### 7 Environment variables

The parameter system uses the following environment variables:

- **SAS_PATH**: Parameter specification files are located by searching the 'config' directory of each path specified by the SAS_PATH environment variable.
- **SAS_CLMODE**: This environment variable may be used to select the command-line syntax:
  - **SAS** - Normal SAS mode
  - **PCS** - Special mode for PCS tools

The default mode is **SAS**.

#### 7.1 PCS mode

The parameter system includes support for an alternative command-line syntax, required by the PCS tools. This mode is selected using the following environment variable:

```
setenv SAS_CLMODE PCS
```

In this mode, only the parameter names CCF, SF, COEFF and ODF should be specified. These options should have a single minus sign and be followed by a space-separated list of arguments. Arguments before the first of these flags are assumed to be ODF items.

For example, the following PCS command line:

```
foo odf1 odf2 -COEFF co1 co2 -CCF ccf1 ccf2 -SP res
```

is equivalent to the following SAS command line:

```
foo --ODF="odf1 odf2" --COEFF="co1 co2" --CCF="ccf1 ccf2" --SP=res
```

PCS mode should not be used for normal SAS tools. PCS mode is a deprecated feature, which may disappear if later releases of the PCS tools use the normal SAS format.
8 Developer’s notes

8.1 Parameter specification file

Every task has an associated parameter specification file, which defines at least the name and type of each parameter. Other optional information includes a default value, allowable range and a prompt string.

The file format is based on the Extensible Markup Language (XML), and the specific elements allowed are described in this section.

8.1.1 XML: quick introduction

The file format is based on the Extensible Markup Language (XML). The syntax looks like the Hyper Text Markup Language (HTML) but, among others, it has strict rules:

- Elements are case sensitive.
- Container elements always require a start tag as well as an end tag.
- Empty XML elements require a slash before the right angle bracket (i.e. <example/>).
- Elements treat whitespace as part of their content.

A quick-and-dirty introduction of the syntax:

**tags** The angle brackets (<> and the names they enclose are called *tags*.

**elements** Elements are the building blocks of a document. An element can be empty or it can be a container of text and/or other elements:

```
<name attr1="value1" attr2="value2" />
<name attr1="value1" attr2="value2">
</name>
```

The starting tag (or single tag if the element is empty) may contain zero or more attributes.

**attributes** An attribute in the tag must have a name and a value assigned to it. The value must always be specified in quotes.

**comments** Comments are elements that are ignored while parsing the document, and are enclosed by <!-- and -->.

```
<!-- This is a comment -->
<text>This is the <bold>new</bold> text</text>
<!-- And this...  
    <text>This was the <bold>obsolete</bold> text</text>
... up to here -->
```

**root element** A document contains one and one only root element.

An example:
8.1.2 XML: escape sequences

In text and attribute values, you need to escape ASCII characters like the angle brackets (&lt;>). Most common encodings are:

" &quot;  
< &lt;  
> &gt;  
& &amp;

For example, the text:

"Me, Myself &lt;I&gt;"

should be encoded in XML as:

&quot;Me, myself &amp; &lt;I&gt;&quot;

Alternatively, you may escape the full string by enclosing it with the CDATA (character data) section. A CDATA section begins with the nine-character delimiter &lt;![CDATA[ and ends with the delimiter ]]&gt;:

&lt;![CDATA["Me, Myself &lt;I&gt;"]&gt;

The only thing that can not go inside the CDATA section is the end-delimiter itself.
8.1.3 Elements

This section describes all elements within a parameter file. The general layout of the parameter file looks like this:

```xml
<!-- root element -->
<FILE>
  <!-- config section -->
  <CONFIG>
    <!-- simple parameter definition -->
    <PARAM id="id1" type="type">
      <DESCRIPTION>text</DESCRIPTION>
      <CONSTRAINTS>expression</CONSTRAINTS>
    </PARAM>
    <!-- nested parameter definition -->
    <PARAM id="id2" type="bool|string" default="value1">
      <DESCRIPTION>text</DESCRIPTION>
      <CASE>
        <ITEM value="value1">
          <PARAM id="id3" type="type"/>
          <PARAM id="id4" type="type"/>
        </ITEM>
        <ITEM value="value2"/>
      </CASE>
    </PARAM>
  </CONFIG>
</FILE>
```

8.1.3.1 CASE

**parent:** PARAM

**children:** list of ITEM elements

**attributes:** none

A CASE element must contain a list of all possible values for this parameter. Currently, only parameters of type bool or string are allowed to contain a CASE element. If you do not specify any default value, the last ITEM found in the list is used as default value.

```xml
<PARAM id="color" type="string" default="red">
  <CASE>
    <ITEM value="red"/>
    <ITEM value="green"/>
    <ITEM value="blue"/>
  </CASE>
</PARAM>
```

Within a PARAM definition, you should not specify both CASE and CONSTRAINTS at the same time.
8.1.3.2  CONFIG

**parent:** FILE

**children:** list of PARAM elements

**attributes:** none

The CONFIG element contains all parameter definitions.

8.1.3.3  CONSTRAINTS

**parent:** PARAM

**children:** selectlib expression

**attributes:** none

The optional CONSTRAINTS element allows to set extra conditions for a particular type. These conditions are expressed in selectlib syntax, and this expression should evaluate to true or false. Currently an expression may only contain a reference to the parameter itself.

If you do not specify a CONSTRAINTS element, the system will insert true as the expression.

The following examples all have exactly the same constraints, utilising the flexibility of the selectlib syntax. Please note, that expressions may contain characters that need to be escaped in XML.

```
<PARAM id="radius" type="int" default="2">
<!-- selectlib using the in-operator -->
<CONSTRAINTS>
  radius in (0:100)
</CONSTRAINTS>
</PARAM>

<PARAM id="radius" type="int" default="2">
<!-- selectlib using the C++ operators escaped in XML -->
<CONSTRAINTS>
  radius &gt; 0 &amp;&amp; radius &lt;= 100
</CONSTRAINTS>
</PARAM>

<PARAM id="radius" type="int" default="2">
<!-- selectlib using the C++ operators within a XML-CDATA section -->
<CONSTRAINTS>
  <![CDATA[ radius &gt; 0 &amp;&amp; radius &lt;= 100 ]]>  
</CONSTRAINTS>
</PARAM>
```
Within a PARAM definition, you should not specify both CASE and CONSTRAINTS at the same time.

8.1.3.4 DESCRIPTION

parent: PARAM

children: text

attributes: none

DESCRIPTION is optional element that describes what this parameter is used for. Though it is optional, developers are encouraged to use it, as it allows the infrastructure to generate help messages for the user.

8.1.3.5 FILE

parent: none, as it is the root element

children: CONFIG

attributes: none

The FILE element currently contains a single CONFIG element only.

8.1.3.6 ITEM

parent: CASE

children: list of PARAM elements

attributes: value required unique and valid (in terms of type) value of the PARAM to which it belongs to.

Depending on the value of the parameter, it may bring other parameters into scope.
<PARAM id="color" type="string" default="blue">
  <CASE>
    <ITEM value="red"> <PARAM id="hell" type="bool" default="yes"/>
      </ITEM>
    <ITEM value="green"/>
    <ITEM value="blue"/>
  </CASE>
</PARAM>

8.1.3.7 PARAM

parent: CONFIG, ITEM

children: CASE, CONSTRAINTS, DESCRIPTION

attributes:  
id required  unique identifier of this parameter
  Each parameter has an id which is used to reference it within the task code. For the
  XMM SAS, ids should comply with the project naming conventions, which means that
  the id should be written in lowercase letters and may contain digits and underscores
  with the following exceptions:
  - mode is a reserved word. This id has a special significance for FTOOLS, where it
    controls the mode of other parameters. Though this behaviour is not implemented
    in the SAS, the SAS may trigger FTOOLS.
  - An id should start with a letter.

type required specification of the parameter type
  list attribute to denote whether the parameter is a list of the specified type or not. Defaults
  to "no".

mandatory describes whether the parameter should be given a value on the command line or not.
  Defaults to "no".

default describes the default value of a parameter. If none given, it defaults to anything
  reasonable for that type. If this element has a child CASE or CONSTRAINTS element,
  you have to make sure that the default is valid for those elements.

A few notes on the child elements:

- All children of this element are optional. We encourage the developer to use at least the
  DESCRIPTION element.
- You should not specify both CASE and CONSTRAINTS.
- Only parameters of type bool and string are currently allowed to use the CASE element.

8.1.4 Command-line behavior

Because of the possibility to nest parameters, you can create a mandatory parameters that may/may not
be specified on the command-line, depending on the value of its parent parameter!

This allows complex constructs such as:
In the following paragraphs, we will use the example above, to explain the behavior with respect to specifying parameters on the command line.

A few notes on the example:

- We did not specify the \textit{mandatory} attribute for parameter \textit{withimage}, hence it defaults to \textit{mandatory="no"}
- In order not to pollute the example, we used a short notation for some of the elements.

\subsection{Default}

\verbatimverbatim
> task // ok
\endverbatimverbatim

The attribute \textit{default="no"} of parameter \textit{withimage} makes sure, that the task will not complain about mandatory parameters \textit{x} and \textit{y}

\subsection{Explicit} Same as the default behavior:

\verbatimverbatim
> task withimage=no // ok
\endverbatimverbatim

Same, as last value specified on the command line takes precedence over previously set values:

\verbatimverbatim
> task withimage=yes withimage=no // ok
\endverbatimverbatim

Note that we do not specify \textit{binsize}. This is fine, as it is \textit{optional} anyways:

\verbatimverbatim
> task withimage=yes x=1 y=2 // ok
\endverbatimverbatim

Forgot to specify the mandatory parameter \textit{y}

\verbatimverbatim
> task withimage=yes x=1 // error
\endverbatimverbatim
Mandatory parameter \( x \) is specified, but we explicitly set \( \text{withimage}=\text{no} \), so setting the value of \( x \) was not necessary:

\[
> \text{task } x=1 \ \text{withimage=\text{no}} \ /\ / \ \text{ok}
\]

**8.1.4.3 Implicit** The implicit behavior makes sure, that whenever you set parameter on the command-line, and the parameter belongs to a specific value of a parent parameter, this parent parameter is implicitly set **unless the parent parameter is set explicitly on the command line too**. As a result, this can save the user a lot of typing!

Mandatory parameters \( x, \) and \( y \) are explicitly set, and this implies \( \text{withimage}=\text{yes} \):

\[
> \text{task } x=1 \ y=1 \ /\ / \ \text{ok}
\]

Mandatory parameters \( x \) is explicitly set. This implies that \( \text{withimage}=\text{yes} \). As result the system will complain about mandatory parameter \( y \):

\[
> \text{task } x=1 \ /\ / \ \text{error}
\]

**8.2 Programmer’s interface**

The API of the parameter library provides functions for accessing parameters from both C++ and Fortran-90 tasks.

There are only 4 basic parameter types (plus corresponding list types), as far as access within the task is concerned:

```c
bool
int
real (time,angle)
string (directory,dataset,file,array,...)
```

For example, a file parameter is accessed, within the task, as if it were a simple string parameter. The use of distinct types, in the parameter file, allows the GUI to present the user with an appropriate widget, such as a file dialog.

The parameter interface also converts a variety of formats for time and angle parameters into a single internal representation:

- Times are accessed by the task as real values which represents the number of seconds since the XMM mission reference time.
- Angles are accessed by the task as a real value in degrees.
8.2.1 C++ API

The C++ API provides functions for accessing the parameters passed to tasks. The parameter interface takes care of error handling, including range checking for certain types.

8.2.1.1 Access parameters  The following functions are provided to access the basic parameter types (that is: bool, int, real and string).
Parameters of a list type are accessed by specifying the index as an additional argument:

```c++
bool    boolParameter(const string& name[], int pos);
int     intParameter(const string& name[], int pos);
double  realParameter(const string& name[], int pos);
string  stringParameter(const string& name[], int pos);
```

The number of elements in a list can be queried by the following function:

```c++
int    parameterCount(const string& name);
```

Note that the index of the first item is zero. The number of elements in a list may be zero.

List parameters may also be accessed as STL vectors using the following functions:

```c++
vector<bool>   boolListParameter(const string& name);
vector<int>    intListParameter(const string& name);
vector<double> realListParameter(const string& name);
vector<string> stringListParameter(const string& name);
```

Note: boolListParameter currently returns vector<int>, instead of vector<bool>, as a temporary work-around for a problem.

8.2.1.2 Setting parameters  It is also possible to set the value of a parameter (which must already be declared in the parameter file) using the functions:

```c++
void    setParameter(const string& name, TYPE value);
```

Where TYPE is one of:

```c++
int
double
const string&
const char*
const vector<bool>&
const vector<int>&
const vector<double>&
const vector<string>&
```
Overloaded versions are also available, in which the ”name” argument is a ”const char*”. For example:

    void setParameter(const char* name, bool value)

Note: setParameter currently requires a vector<int> to be used for a boolean list, as a temporary work-around for a problem.

8.2.1.3 Example  Consider the following parameter specification file:

    <PARAM id="bval" type="bool" default="no"/>
    <PARAM id="ival" type="int"/>
    <PARAM id="rval" type="real"/>
    <PARAM id="sval" type="string" default="ab cd"/>
    <PARAM id="rlist" type="real" list="yes"/>

These parameters may be read within a C++ task, as follows:

    // include headers
    #include <param.h>
    #include <string>
    #include <vector>

    // access parameters
    bool bval = booleanParameter("bval");
    int ival = intParameter("ival");
    double rval = realParameter("rval");
    string sval = stringParameter("sval");
    vector<double> rlist = realParameterList("rlist");

    // The parameter list "rlist" may also be read as follows:
    vector<double> rlist;
    int n = parameterCount("rlist");
    for(int i=0; i<n; i++)
        rlist.push_back(realParameter("rlist",i));

8.2.2 Fortran-90 API

The Fortran-90 API provides functions for accessing the parameters passed to tasks. The parameter interface takes care of error handling, including range checking for certain types.

8.2.2.1 Access parameters  The following functions are provided to access the basic parameter types (that is: bool, int, real and string).
Parameters of a list type are accessed by specifying the index as an additional argument:

    function booleanParameter(name,index)
        character(len=*) , intent(in) :: name
        integer, optional, intent(in) :: index
end function booleanParameter

function intParameter(name,index)
   character(len=*) , intent(in) :: name
   integer , optional, intent(in) :: index
end function intParameter

function realParameter(name,index)
   character(len=*) , intent(in) :: name
   integer , optional, intent(in) :: index
end function realParameter

function stringParameter(name,index)
   character(len=*) , intent(in) :: name
   integer , optional, intent(in) :: index
end function stringParameter

The stringParameter function is used for both string and file-name parameters.

The number of elements in a list can be queried by the following function:

function parameterCount(name)
   character(len=*) , intent(in) :: name
   integer :: count
end function parameterCount

Note that the index of the first item is zero. The number of elements in a list may be zero.

8.2.2.2 Setting parameters It is also possible to set the value of a parameter (which must already
be declared in the parameter file) using the interface:

subroutine setParameter(name,value)
   character(len=*) , intent(in) :: name
   TYPE , intent(in) :: value
end subroutine

Where TYPE is one of:

   integer(kind=int32)
   integer(kind=int32), dimension(:)
   real(kind=single)
   real(kind=single), dimension(:)
   real(kind=double)
   real(kind=double), dimension(:)
   character(len=*)
   character(len=*) , dimension(:)

8.2.2.3 Example Consider the following parameter specification file:
These parameters may be read within a Fortran-90 task, as follows:

```fortran
use param

logical :: bval
integer :: ival, n
real :: rval
character(len=128) :: sval

bval = booleanParameter("bval")
ival = intParameter("ival")
rval = realParameter("rval")
sval = stringParameter("sval")

n = parameterCount("rlist")
do i = 0, n-1
   rlist(i) = realParameter("rlist",i)
end do
```

It is useful to remove trailing spaces returned by stringParameter, using the Fortran-90 'trim' function.

### 8.3 Conditional parameters

The GUI supports the notion of conditional parameters, which are enabled by certain values of other parameters. These makes it clear to the user when particular parameters are relevant. The enabling of parameters in the GUI is controlled by an (optional) layout file. Reference should be made to the GUI documentation for further details.

It is important that the task code only uses optional parameters when the corresponding option is enabled. It is recommended that the task reads in all parameters at the beginning, in a single section of code. This makes it easier to check that nested use of conditional parameters matches that in the layout file.

#### 8.3.1 Optional processing

When part of the processing performed by a task is optional (e.g. an instrumental correction), the corresponding parameters are only used when the option is selected. A boolean parameter should be used to enable the processing in the task GUI.

For example, consider an optional smoothing operation which requires two parameters, alpha and beta:

```xml
<!-- parameter file snippet begin -->
<PARAM id="smooth" type="bool" default="no"/>
```
The GUI groups the parameters ‘alpha’ and ‘beta’ in a frame, labelled ‘smooth’. The parameters in the frame are enabled by the boolean parameter ‘smooth’.

8.3.2 Alternative processing

A task may provide alternative processing options, each of which has its own set of parameters. The GUI allows the user to select one of the options from a pull-down menu. Only the parameters relevant to the selected option are displayed.

The following example provides two smoothing algorithms, called ‘smart’ and ‘simple’. A pop-up menu provides the options ‘smart’, ‘simple’ and ‘none’. The latter performs no smoothing.

```xml
<!-- parameter file snippet begin -->
<PARAM id="smooth" type="string" default="none"/>
</ITEM>
<ITEM value="none"/>
<ITEM value="simple"/>
  <PARAM id="factor" type="real" default="3."/>
</ITEM>
<ITEM value="smart">
  <PARAM id="alpha" type="real" default="0"/>
  <PARAM id="beta" type="real" default="0"/>
</ITEM>
</ITEM>
</CASE>
</PARAM>
<!-- snippet end -->

// C++ Task
string mode = stringParameter("smooth");
if(mode == "smart") {
    alpha = realParameter("alpha");
    beta = realParameter("beta");
    ... // Process in smart mode
} else if(mode == "simple") {
    factor = realParameter("factor");
    ... // Process in simple mode
```
} else if (mode == "none") {
    ... // Do nothing
} else {
    errmsg << "Unrecognised option " << mode
    << " for parameter smooth" << error("paramAlternative");
}

If 'smart' is selected on the menu, the 'smooth' box contains the 'alpha' and 'beta' widgets, associated
with this option. If 'simple' is selected, the box contains the 'alpha' and 'gamma' widgets. If 'none' is
selected, then the box is disabled.

8.4 Accessing datasets

Special parameter types are available for tables, arrays, blocks and columns.

These allow a dataset component, such as a column and its associated table and dataset, to be specified
by a single parameter of the form 'set:table:column'.

The DAL functions accept these colon-separated strings, as arguments, and extract the appropriate field.
Hence, the whole string can be passed to each of the DAL functions.

For example, in a Fortran-90 task:

    ! Read array
    arrayname = stringParameter("image")
    set1 = dataSet(arrayname, READ)
    arr = array(set1,arrayname)

    ! Read column
    colname = stringParameter("column")
    set2 = dataSet(colname, READ)
    tab = table(set2,colname)
    col = column(tab,colname,Read)

Suitable parameter file entries would be:

    <!-- parameter file snippet begin -->
    <PARAM id="image" type="array"/>
    <PARAM id="column" type="column"/>
    <!-- snippet end -->

8.5 Design

These notes explain a few aspects of the design of the SAS parameter interface.

* The various parameter types are represented by subclasses of the abstract class Parameter.
  Each parameter class has a corresponding widget class in the GUI.
- The Parameter class implements an MVC mechanism, that allows a view (e.g. GUI widget) to receive notification when a parameter value changes. Views implement the Parameter-View interface and register themselves with a Parameter model. This approach cleanly decouples the parameter model from the view.

- The parsing of the parameter file is decoupled from the implementation of the parameters, by using separate Reader classes which implement the ParamInfoReader interface. The '.par' files are parsed by an XmlInfoReader.

  This design allows the possibility to plug-in alternative readers (note that the former implementation used an XpiInfoReader) to handle other file formats. The appropriate reader could be chosen according to the file-name extension (e.g. '.par') of the parameter file.

- Separate files are used for the parameter definitions ('.par') and GUI layout information ('.lyt'). This decouples the concerns of parameter definitions and their graphical representation; the parameter interface may be used without the GUI, from the command line. This also allows layout files to be used with existing FTOOLS.

- SAS errors are only generated by top-level wrapper functions, which implement the task API. The underlying implementation simply propagates an error status as the return values from nested calls. This approach allows the GUI to recover gracefully from parameter errors, without crashing.

8.6 To-do list

The following enhancements to the parameter interface are nice to have:

- Export only public interfaces
- Support globbing of file-name-list parameters: -foo='*.fits'
- Support in—out—inout attributes on file, dataset etc... parameters
- Add check that parameters are not defined more than once in .par file
- Add parameter referencing, such that a parameter can occur in multiple parents
- Escape single quote when printing command line
- Check implementation for memory leaks
- printing all parameters should show parent-child relations

References