Australian Government
Department of Defence
Defence Science and
Technology Organisation

The Janus C++ Library - An Interface Class for DAVE-ML Compliant XML-Based Flight Model Datasets.

Dr D. M. Newman
(on contract from Ball Solutions Group)

Air Vehicles Division
Defence Science and Technology Organisation

ABSTRACT

The Australian Defence Science and Technology Organisation's (DSTO) Flight Systems Branch undertook a review of its aircraft flight model development processess in 2003. As an outcome from the review it was decided to align future flight model dataset strucutres with the American Institute of Aeronautics and Astronautics (AIAA) draft modelling and simulation standard [AIAA, 2003] and the related Dynamic Aerospace Vehicle Exchange Markup Language (DAVE-ML). Ball Solutions Group were contracted to develop an application programming interface library to the DAVE-ML datset structure for use with flight dynamic simulation and performance estimation models. The library is implemented in C++ and is known as 'Janus'. The Janus library is detailed in this report.

RELEASE LIMITATION

Approved for public release

Published by

DSTO
506 Lorimer St
Fishermans Bend, Victoria 3207   Australia

Telephone: (03) 9626 7000
Fax: (03) 9626 7999

August 2005

THE JANUS C++ LIBRARY – AN INTERFACE CLASS FOR DAVE-ML COMPLIANT XML-BASED FLIGHT MODEL DATASETS

Document Number :	31495.002
Revision Number :	Version 1.00
Document Date :	18 July 2005

SOLUTIONSGROUP
The Way Ahead

AUTHORISATION

This document has been prepared by Ball Solutions Group, ABN 66 072 963 690 for the Defence Science and Technology Organisation, Flight Systems Branch of Air Vehicles Division, in accordance with Attachment A to Contracts No. 123394 and 139826.

D. M. Newman
Senior Aerospace and Flight Test Engineer
Ball Solutions Group

18 July 2005

DOCUMENT REVISION HISTORY

Version	Effect	Date
0.90	Original Draft Release	23 July 2004
0.91	additional function documentation	15 October 2004
0.92	additional function documentation, including simple code examples	14 December 2004
0.93	modified API to include 3 types of output variables, replaced dependent variable access procedures with output variable access	22 December 04
0.94	multi-dimensional polynomial interpolation added	11 January 2005
0.95	ungridded interpolation added	7 February 2005
0.96	basic MathML logic capability added	11 February 2005
0.97	namespaces added	12 March 2005
0.98	file header processing and output	29 March 2005
0.99	string array handling added	30 June 2005
1.00	encrypted XML handling added	18 July 2005

Contents

List of Abbreviations

ADF	Australian Defence Force
AIAA	American Institute of Aeronautics and Astronautics
AVD	Air Vehicles Division
DAVE-ML	Dynamic Aerospace Vehicle Exchange Markup Language
DoF	degree of freedom
DOM	Document Object Model
DSTO	Defence Science and Technology Organisation
DTD	Document Type Description
FS	Flight Systems Branch
MathML	Mathematical Markup Language
NaN	Not-a-Number
SI	Systeme International d'Unites
XML	eXtensible Markup Language

1 Introduction

1.1 Background

Defence Science and Technology Organisation (DSTO) Flight Systems Branch has reviewed its flight model development and maintenance processes, in conjunction with the requirements of Defence flight model users [Newman, 2004], and has decided to align its future flight model dataset structures with the American Institute of Aeronautics and Astronautics (AIAA) draft modelling and simulation standard [AIAA, 2003] and the related Dynamic Aerospace Vehicle Exchange Markup Language (DAVE-ML) Document Type Description (DTD) [Jackson & Hildreth, 2002], [AIAA, 2004]. Ball Solutions Group has been contracted to develop a programming library, providing an application programming interface to the DAVE-ML dataset structure, for use by DSTO Flight Systems in future development of performance, flight dynamic and other aircraft models. The library is implemented as a C++ class known as 'Janus.'

1.2 Purpose

This document outlines the usage and structure of the Janus C/C++ programming library which is under development for DSTO Flight Systems Branch by Ball Solutions Group.

1.3 Scope

The elements of the Janus library described in this document are:

a. Methods of instancing the Janus class,

b. The related JanusErr class, which provides information related to library initialisation problems,

c. Public types in the classes,

d. Public functions in the classes, and

e. Source file components applicable to each function.

1.4 Overview

The library provides a flight modelling programmer with direct access to an eXtensible Markup Language (XML) dataset which conforms to the draft AIAA modelling standard as implemented under the DAVE-ML DTD version 1.7b1 [AIAA, 2004], in the form of a C++ class. It was developed under Linux using the gcc 3.3.4 compiler, and has received limited testing under Microsoft Windows 2000 using the Visual C++ compiler and under Microsoft Windows XP using the cygwin environment and the gcc 3.3.1 compiler. To load and parse an XML file, the Janus class makes use of the Apache Xerces-C++ validating XML parser library, currently at version 2.5.0.

When initialised, which requires the calling program to supply an XML dataset file name, the library creates and loads a Document Object Model (DOM) from the file, using the Xerces-C++ parser, then extracts numerical data from the DOM and stores it in a arrays for access from the calling program through the Janus interface. Depending on the dataset and the application, further initialisation may be required after first instantiation. During the initialisation process, problems with the XML file or its contents will cause a JanusErr to be thrown, providing a relevant message. If the calling program does not catch this error, execution will abort.

With initialisation complete, the calling program can supply the current state values of relevant independent variables through the Janus interface and receive in return a single output variable value compatible with the independent variables. The forms of the data and the interpolation, curve fitting or function evaluation required to generate the dependent variable value are controlled by the XML data file content and are transparent to both the calling program and the user. For these functions, which may be called repeatedly during program execution, speed of execution is a priority and so limited error checking or notification is performed.

Errors from the post-initialisation functions of the Janus instance therefore have the potential to cause aborts without the option of handling by the calling program. However, the returned values may be used to determine error conditions, since errors in functions returning double precision data always return Not-a-Number (NaN), errors in functions returning string data always return a zero pointer, errors in functions returning integer data or flags always return -1, and errors in functions returning enums return an ERROR[*] value.

At any stage after initialisation, the Janus instance may be queried for details of any variable or function, including units, names, descriptions, minima, maxima, and interpolation or extrapolation attributes.

1.5 Data Types

To the modeller whose code uses a Janus instance to determine variable values, the underlying form of the XML dataset is irrelevant. However, the dataset developer needs to take account not just of the DAVE-ML DTD which guides production of a well-formed, valid dataset, but also of how the Janus treats each data type. The three main data types which will be encountered are:

a. Gridded data, arranged in up to 32 dimensions on a regular grid, which can be interpolated or extrapolated either linearly or using polynomials of order zero (nearest) to three inclusive;

b. Ungridded data, a cloud of arbitrarily located data points forming a convex hull, which is partitioned using Delaunay triangulation and interpolated multi-linearly; and

c. Functional representation in Mathematical Markup Language (MathML) form, which is evaluated in accordance with the mathematical operators shown in the dataset. At present Janus implements only the more common operators and qualifiers defined in the MathML DTD. It deals only with real number data, but includes logical operators which return Boolean qualifiers within a calculation element.

For every dataset to be accessed, there will be a preferred data type based on the form of the data and its possible applications. In choosing how to represent a particular piece of data within the XML dataset, the modeller should consider how to best make use of Janus's capabilities. Where relevant to computational comparisons below, the software is considered as running on a 'typical' engineering-use PC circa 2005, under either Windows + Cygwin or Linux. Some aspects which may be relevant are:

a. Gridded data using linear interpolation is generally the fastest to evaluate, with Janus performing several million evaluations per second on a representative aerodynamic dataset. As the number of degrees of freedom is increased for datasets of equivalent complexity, function evaluation speed typically halves for each additional degree of freedom.

b. Polynomial interpolation of gridded data is typically several times slower than linear interpolation of the same data.

c. Ungridded data is generally the slowest to evaluate. For one degree of freedom (DoF) data, an ungridded interpolation will typically take four or five times as long as gridded interpolation of the same data based on the same breakpoints. This is because of the added complexity of the barycentric coordinate computation used to weight the contributing data points. In addition, as the number of degrees of freedom is increased for datasets of equivalent complexity, function evaluation speed typically reduces by an order of magnitude for each additional degree of freedom.

d. Extrapolation of any form of data is inherently risky. If you choose to take the risk, gridded data extrapolation is much safer than ungridded data extrapolation. Because the ungridded data is processed in barycentric coordinates, not cartesian coordinates, and checking cartesian directions wastes processing time, Janus will only extrapolate such data if *all* independent variables of the function are set to be extrapolated in both directions.

e. MathML functions are evaluated at speeds similar to gridded data of equivalent complexity. However, high order polynomial evaluation can be computationally costly. Luckily, high order polynomials are almost always a bad choice for representation of aeronautical data.

f. An extension to the DAVE-ML standard allows arrays of strings to be stored in and accessed from gridded tables. The applications of this are quite limited, and the related function documentation should be fully complied with for successful use.

2 Janus Module Documentation

2.1 Class Instantiation

Enumerations

- enum Janus::VersionType { Janus::SHORT, Janus::LONG }

Functions

- char * Janus::getJanusVersion (const VersionType versionType)
- char * Janus::getXmlFileName ()
- Janus::Janus (const char *documentName, const bool validate)
- Janus::Janus (const char *documentName)
- Janus::Janus ()
- int Janus::setDomValidation (const bool validate)
- int Janus::setXmlFileName (const char *documentName)
- int Janus::writeXmlFile (const char *fileName)
- Janus::~Janus ()

2.1.1 Detailed Description

The initialisation functions relate to the construction and destruction of a Janus instance. They perform XML initialisation and instance an Xerces parser, then load from the supplied XML file to a DOM structure and create numeric arrays based on the XML data.

The early stages of the initialisation process will throw JanusErr exceptions if the XML file is not found or does not load or parse successfully. If the calling program does not catch these exceptions, the program will abort. An example of exception handling, applicable to all forms of Janus initialisation, is:

```
int iflag;
try {
  iflag = prop.setXmlFileName( fileName );
}
catch ( const JanusErr &excep ) {
  cerr << excep.type << " \n\n";
  return 1;
}
```

2.1.2 Enumeration Type Documentation

2.1.2.1 enum VersionType `[inherited]`

This enum is used to indicate whether a short or long library version description string is required.

Enumerator:

SHORT a short, purely numeric string, eg "0.97"

LONG a longer, alpha-numeric string, eg "Janus V-0.97"

2.1.3 Function Documentation

2.1.3.1 char * getJanusVersion (const VersionType *versionType*) [inherited]

This function allows the calling program to determine the version of the Janus library which is in use. It is particularly useful for dynamically linked programs which may be used with several different library versions.

Parameters:

versionType determines whether a short or long string is returned.

Returns:

a pointer to the version description string.

2.1.3.2 char * getXmlFileName () [inherited]

If the instance has been fully initialised, the fully-qualified name of the XML dataset file from which it was initialised is returned by this function.

Returns:

The XML file name e.g. "~/pika/pika_prop.xml"

2.1.3.3 Janus (const char * *documentName*, const bool *validate*) [inherited]

The constructor may be called with the XML document name and an instruction regarding validation of the DOM against the DTD. Setting `validate` to *false* in this instantiation will cause the DTD to be entirely ignored. The default, if the `validate` parameter is not supplied, is to validate the XML document.

Parameters:

documentName is the XML file name, e.g. "~/pika/pika_prop.xml"

validate options, controlling the DOM relationship with the DTD, are: [*true* | *false*]

See also:

setDomValidation

2.1.3.4 Janus (const char * *documentName*) [inherited]

The constructor, called with the XML document name, does the XML utilities and Xerces parser initialisation, then uses them to load the DOM structure. A minimal example is:

```
#include <string>
#include "Janus.h"

using namespace std;

int main (int argc, char* args[])
{
  char fileName[] = "~/pika/pika_prop.xml";
  Janus prop( fileName );

  return 0;
}
```

When the XML file name is supplied, either at instantiation or afterwards when using setXml-FileName, the DOM is parsed and validated against *DAVEfunc.dtd* before pointers and data arrays are set up. The constructor calls private functions within the class to set up pointers to *variableDef*, *breakpointDef*, *griddedTableDef* and *function* DOM Level 1 elements, and to set up arrays of breakpoint and function table values extracted from the DOM.

Lastly it allocates an array for the values of all variables referenced in the DOM. While the instance remains in scope, this array maintains the most recent value for each variable, set by either the calling program (for independent variables) or by the function evaluation within Janus (for dependent variables). If an initial value for a variable has been supplied in the XML file, it is placed in this array during initialisation. Other variables are set to zero during initialisation.

Parameters:

documentName is the XML file name, e.g. "~/pika/pika_prop.xml"

2.1.3.5 Janus () `[inherited]`

The constructor can be used to instance the Janus class without supplying a name for the XML file from which the DOM is constructed, but in this state is not useful for any class functions. It will require to be supplied with the XML file name before any further use of the instanced class.

This form of the constructor is principally for use within higher level instances, where memory needs to be allocated before the data to fill it is specified.

See also:

setXmlFileName

2.1.3.6 int setDomValidation (const bool *validate*) `[inherited]`

This function allows initial validation of the DOM against the DTD to be turned off or on. The Janus library defaults to not validating the DOM. Use of this function can require validating the XML data structure using the DTD specified in the XML file header, which will lead to an error during Janus instantiation if the XML file is invalid. Note that ∗∗∗ at present ∗∗∗ the DTD is always loaded, whether validation is required or not, because it is necessary to define a MathML namespace tag. This function has no effect if called after the XML file name is passed to the instance.

Parameters:

validate options, controlling the DOM relationship with the DTD, are: [*true* | *false*]

Returns:

0 if internal flag set successfully

2.1.3.7 int setXmlFileName (const char ∗ *documentName*) `[inherited]`

An uninitialised instance of Janus is associated with a particular XML file by this function. The DOM within the Janus instance is loaded from the named file, parsed, and optionally validated against *DAVEfunc.dtd* before pointers and data arrays are set up. The behaviour if another XML file name is supplied to an instance which has already been initialised is undefined, but a check is performed at load time which results in a JanusErr exception under these circumstances.

Parameters:

documentName is the XML file name, e.g. "~/pika/pika_prop.xml"

Returns:

0 if initialisation completes successfully

See also:

JanusErr
setDomValidation

2.1.3.8 int writeXmlFile (const char * *fileName*) [inherited]

This function allows the calling program to duplicate the contents of the DOM as currently loaded, in an XML text file. Failures during the process may throw XML exceptions or DOM exceptions.

Parameters:

fileName is a legal, fully qualified file name to which the current contents of the DOM will be written in XML format.

Returns:

A value of 0 is returned if the XML file is written successfully.

2.1.3.9 ~Janus () [inherited]

After deleting the array allocations for function definitions, gridded table definitions, breakpoint definitions and variable definitions, in that order, the parser instance is released and the last act of the destructor is to terminate the platform utilities. The destructor is called automatically when the instance goes out of scope.

2.2 XML File Documentation

Enumerations

- enum Janus::AuthorAttribute {
  Janus::NAME, Janus::ORG, Janus::XNS, Janus::EMAIL,
  Janus::ADDRESS }
- enum Janus::ModificationAttribute {
  Janus::MODID, Janus::MOD_REFID, Janus::MOD_AUTHORNAME, Janus::MOD_-AUTHORORG,
  Janus::MOD_AUTHORXNS, Janus::MOD_AUTHOREMAIL, Janus::MOD_-AUTHORADDRESS, Janus::MOD_DESCRIPTION }
- enum Janus::ReferenceAttribute {
  Janus::REFID, Janus::AUTHOR, Janus::TITLE, Janus::ACCESSION,
  Janus::DATE, Janus::HREF, Janus::DESCRIPTION }

Functions

- char ∗ Janus::getXmlFileAuthor (const AuthorAttribute authorAttribute)
- char ∗ Janus::getXmlFileCreationDate ()
- char ∗ Janus::getXmlFileDescription ()
- char ∗ Janus::getXmlFileModification (const int index, const ModificationAttribute modificationAttribute)
- int Janus::getXmlFileModificationCount ()
- int Janus::getXmlFileModificationExtraDocCount (const int index)
- char ∗ Janus::getXmlFileModificationExtraDocRefID (const int index, const int indxRef)
- char ∗ Janus::getXmlFileReference (const int index, const ReferenceAttribute referenceAttribute)
- int Janus::getXmlFileReferenceCount ()
- int Janus::getXmlFileReferenceIndex (const char ∗refID)
- char ∗ Janus::getXmlFileVersion ()

2.2.1 Detailed Description

The documentation functions relate to the descriptive material contained in the XML dataset file header. This includes file authorship, modification records, and cross-references to source material. The functions provide access to this data for the calling program. Some of this reference material is optional, as defined by the DAVE-ML DTD, so many of these functions may return blank strings if requested data is not present.

2.2.2 Enumeration Type Documentation

2.2.2.1 enum AuthorAttribute `[inherited]`

This enum serves as input to getXmlFileAuthor, and is used to indicate which of the XML dataset file's author attributes is required.

Enumerator:

NAME The author's name

ORG The organisation directing the author's work on this dataset

XNS Extensible Name Service (XNS) is an open XML-based protocol that specifies a way to establish and manage a universal addressing system. An XNS universal address serves as a permanent contact point for an individual or other legal entity, such as a business. This XNS entry provides contact details for the author or his organisation in reference to this dataset.

EMAIL The e-mail address through which the author may be contacted in reference to this dataset

ADDRESS The postal mail address through which the author may be contacted in reference to this dataset

2.2.2.2 enum ModificationAttribute `[inherited]`

This enum serves as input to getXmlFileModification, and is used to indicate which of the XML dataset file's *modificationRecord* attributes is required. Not all datasets will contain all attributes. As defined in DAVEfunc.dtd, only the modificationID and the author base data are required.

Enumerator:

MODID The modificationRecord ID

MOD_REFID The reference ID on which the modification is based

MOD_AUTHORNAME The name of the modification's author

MOD_AUTHORORG The organisation directing the author's work on this modification of the dataset

MOD_AUTHORXNS Extensible Name Service (XNS) is an open XML-based protocol that specifies a way to establish and manage a universal addressing system. An XNS universal address serves as a permanent contact point for an individual or other legal entity, such as a business. This XNS entry provides contact details for the author or his organisation in reference to this modification of the dataset.

MOD_AUTHOREMAIL The e-mail address through which the author may be contacted in reference to this modification of the dataset

MOD_AUTHORADDRESS The postal mail address through which the author may be contacted in reference to this modification of the dataset

MOD_DESCRIPTION A description of the modification

2.2.2.3 enum ReferenceAttribute `[inherited]`

This enum serves as input to getXmlFileReference, and is used to indicate which of the XML dataset file's *reference* attributes is required. Not all datasets will contain all attributes. As defined in DAVEfunc.dtd, description, accession number and href are optional.

Enumerator:

REFID The reference ID

AUTHOR The reference document's author name

TITLE The reference document's title

ACCESSION The reference document's library accession number

DATE The date of publication of the reference document

HREF The XLink address or identifier of the reference document

DESCRIPTION A description of the reference document

2.2.3 Function Documentation

2.2.3.1 char * getXmlFileAuthor (const AuthorAttribute *authorAttribute*) [inherited]

This function provides access to the *author* attribute character strings contained in the XML dataset file header. Some attributes (described in AuthorAttribute) are optional.

Parameters:
authorAttribute indicates which of the available author attributes is required by this function call.

Returns:
A pointer to the requested attribute is returned. If an optional attribute is not present a blank string is returned.

2.2.3.2 char * getXmlFileCreationDate () [inherited]

This function provides access to the *fileCreationDate* character string contained in the XML dataset file header. The format of the dataset string is determined by the XML dataset builder, but DAVE-ML recommends the ISO 8601 form "2004-01-02" to refer to 2 January 2004.

Returns:
a pointer to the XML file creation date string.

2.2.3.3 char * getXmlFileDescription () [inherited]

This function provides access to the *description* character string contained in the XML dataset file header. The description consists of a string of arbitrary length, which can include tabs and new lines as well as alphanumeric data. This means pretty formatting of the XML source will also appear in the returned description string. Since description of a file is optional, the returned string may be blank.

Returns:
a pointer to the XML description string.

2.2.3.4 char * getXmlFileModification (const int *index*, const ModificationAttribute *modificationAttribute*) [inherited]

This function provides access to the *modificationRecord* attribute character strings contained in the XML dataset file header. Some attributes (described in ModificationAttribute) are optional.

Parameters:
index has a range from 0 to (getXmlFileModificationCount() - 1), and selects the modification record be addressed.

modificationAttribute indicates which of the available *modificationRecord* attributes is required by this function call.

Returns:
A pointer to the requested attribute is returned. If an optional attribute is not present, or a `index` out of range is requested, a blank string is returned.

2.2.3.5 int getXmlFileModificationCount () [inherited]

This function returns the number of *modificationRecord* records at the top level of the *fileHeader* component of the XML dataset.

Returns:
the number of modification records in the XML dataset file header. Possible values are zero or more.

2.2.3.6 int getXmlFileModificationExtraDocCount (const int *index*) [inherited]

As well as its basic `refID` cross-reference, each *modificationRecord* can have extra documents referenced. This function allows the calling program to determine how many, if any, extra document reference records are cross-referenced by each modification.

Parameters:
index has a range from 0 to (getXmlFileModificationCount() - 1), and selects the modification record be addressed.

Returns:
the number of extra documents referenced is returned. If a `index` out of range is requested, -1 is returned.

2.2.3.7 char * getXmlFileModificationExtraDocRefID (const int *index*, const int *indxRef*) [inherited]

Where a *modificationRecord* references extra documents, this function allows the calling program to access the `refID` associated with each of those documents. The result of this function may be used in conjunction with getXmlFileReferenceIndex and getXmlFileReference to obtain the details of the associated reference material.

Parameters:
index has a range from 0 to (getXmlFileModificationCount() - 1), and selects the modification record be addressed.

indxRef has a range from 0 to (getXmlFileModificationExtraDocCount (index) - 1), and selects the extra document record to be addressed.

Returns:
A pointer to the requested `refID` is returned. If a `index` or `indxRef` out of range is requested, a blank string is returned.

2.2.3.8 char * getXmlFileReference (const int *index*, const ReferenceAttribute *referenceAttribute*) [inherited]

This function provides access to the *reference* attribute character strings contained in the XML dataset file header. Some attributes (described in ReferenceAttribute) are optional.

Parameters:
index has a range from 0 to (getXmlFileReferenceCount() - 1), and selects the reference record to be addressed.

referenceAttribute indicates which of the available reference attributes is required by this function call.

Returns:
A pointer to the requested attribute is returned. If an optional attribute is not present, or a `index` out of range is requested, a blank string is returned.

2.2.3.9 int getXmlFileReferenceCount () `[inherited]`

This function returns the number of *reference* records at the top level of the *fileHeader* component of the XML dataset.

Returns:
the number of reference records in the XML dataset file header. Possible values are zero or more.

2.2.3.10 int getXmlFileReferenceIndex (const char * *refID*) `[inherited]`

The file header may contain a list of references. Each of these is associated with a unique reference ID which file modification and data provenance records use for cross-referencing. This function relates the reference ID to an `index` into the arrays of reference data, as used in Janus::getXMLFileReference .

Parameters:
refID is a short, unique string used to refer to elements in the XML file header's list of references.

Returns:
an `index` in the range 0 to (getXmlFileReferenceCount() - 1) is returned. Where the file header contains no reference records, or `refID` does not match any reference records, -1 is returned.

2.2.3.11 char * getXmlFileVersion () `[inherited]`

This function provides access to the *fileVersion* character string contained in the XML dataset file header. The format of the version string is determined by the XML dataset builder. Since the file version is optional in the DAVE-ML DTD, the returned string may be blank.

Returns:
a pointer to the XML file version string.

2.3 XML Tabulated Functions

Functions

- char ∗ Janus::getFunctionDefinitionName (const int index)
- char ∗ Janus::getFunctionDescription (const int index)
- char ∗ Janus::getFunctionName (const int index)
- int Janus::getNumberOfFunctions ()

2.3.1 Detailed Description

These elements of the Janus class provide access to the *function* elements contained in a DOM which complies with the DAVE-ML DTD. Each function has optional description, optional provenance, and either a simple table of input/output values or references to more complete (possibly multiple) input, output, and function data elements. In general, calling programs should access function-based data through the *outputVariable* procedures rather than through these lower-level function access procedures.

All *function* and *dependentVariable* functions use an index based on the *function* Level 1 elements defined in the XML dataset (see Janus::getNumberOfFunctions). For example:

```
int nf = prop.getNumberOfFunctions();
cout << " Number of functions = " << nf << "\n\n";

for ( int i = 0 ; i < nf ; i++ ) {
  cout << "  Function " << i << " : \n"
       << "   Name                     : "
       << prop.getFunctionName( i ) << "\n";
}
```

The order of *function* s within the DOM is arbitrary and the calling program is responsible for determining which `index` addresses each *function*. The function `index` range is from zero to (Janus::getNumberOfFunctions() - 1).

2.3.2 Function Documentation

2.3.2.1 char ∗ getFunctionDefinitionName (const int *index*) `[inherited]`

A function definition's *name* attribute is a string of arbitrary length, but normally short. It is not used for indexing, and therefore need not be unique (although uniqueness may aid both programmer and user), but should comply with the AIAA draft standard [AIAA, 2003] if possible. Note that the *function definition* name is returned, not the *function* name nor the variable ID associated with it. For functions defined in terms of a single list of variable values, there is no explicit function definition and a blank string is returned.

Parameters:

index has a range from 0 to (getNumberOfFunctions() - 1), and selects the function to be addressed.

Returns:

A pointer to an XML *functionDefn* tag's *name* attribute string is returned. The *functionDefn* is a child node of the *function*, so the input `index` refers to the *function*. An `index` out of range will return a zero pointer.

See also:

getFunctionName

2.3.2.2 char ∗ getFunctionDescription (const int *index*) `[inherited]`

A function's description consists of a string of arbitrary length, which can include tabs and new lines as well as alphanumeric data. This means pretty formatting of the XML source will also appear in the returned description string. Since description of a function is optional, the returned string may be blank.

Parameters:

index has a range from 0 to (getNumberOfFunctions() - 1), and selects the function to be addressed.

Returns:

A pointer to an XML *function* tag's *description* child element contents string is returned. An `index` out of range will return a zero pointer.

2.3.2.3 char ∗ getFunctionName (const int *index*) `[inherited]`

A function's *name* attribute is a string of arbitrary length, but normally short. It is not used for indexing, and therefore need not be unique (although uniqueness may aid both programmer and user), but should comply with the AIAA draft standard [AIAA, 2003] if possible. Note that the *function* name is returned, not the function definition name nor the variable ID associated with it.

Parameters:

index has a range from 0 to (getNumberOfFunctions() - 1), and selects the function to be addressed.

Returns:

A pointer to an XML *function* tag's *name* attribute string is returned. An `index` out of range will return a zero pointer.

See also:

getFunctionDefinitionName

2.3.2.4 int getNumberOfFunctions () `[inherited]`

The returned value includes all functions found in the DOM, and makes no distinction between function types. It may be used as the upper limit for an index to address functions and their associated dependent variables, although not all output variables are necessarily associated with functions (e.g. constants and MathML expressions).

Returns:

Total number of all functions defined in the XML file and successfully loaded into the DOM.

2.4 Output Variables

Functions

- int Janus::applyOutputScaleFactorByIndex (const int index, const double factor)
- int Janus::applyOutputScaleFactorByVarID (const char ∗varID, const double factor)
- int Janus::getNumberOfOutputs ()
- double Janus::getOutputScaleFactorByIndex (const int index)
- double Janus::getOutputScaleFactorByVarID (const char ∗varID)
- char ∗ Janus::getOutputVariable (const int index, int)
- double Janus::getOutputVariable (const int index)
- double Janus::getOutputVariableByVarID (const char ∗varID)
- char ∗ Janus::getOutputVariableDescription (const int index)
- char ∗ Janus::getOutputVariableID (const int index)
- int Janus::getOutputVariableIndex (const char ∗varID)
- char ∗ Janus::getOutputVariableName (const int index)
- char ∗ Janus::getOutputVariableUnits (const int index)

2.4.1 Detailed Description

Three types of output variables are defined by the DAVE-ML DTD. They are:

- Dependent variables resulting from *function* evaluation, but not forming an input to another calculation;
- Variables evaluated using MathML, but not forming an input to another calculation; and
- Variables explicitly defined as outputs using the *isOutput* attribute.

These functions provide the means to obtain the characteristics of variables which satisfy these criteria, and to obtain variable values based on the current state of all variables within the Janus instance. Normal usage of the Janus class should rely on these functions for output, since they ensure that returned values are compatible with the current state of all inputs.

2.4.2 Function Documentation

2.4.2.1 int applyOutputScaleFactorByIndex (const int *index*, const double *factor*) `[inherited]`

This function should be used with extreme caution. The default scale factor for each output variable is unity. Each time this function is used, it multiplies by `factor` the current value of scale factor associated with the output variable referenced by `index`. The accumulated scale factor is applied to all subsequent computations used to determine a value for the output variable referenced by `index`. The use of this function is particularly discouraged for datasets where output from one function is defined as input to another function.

Parameters:

index has a range from 0 to (getNumberOfOutputs() - 1), and selects the output variable to be addressed.

factor is the new scale factor to be applied multiplicatively to the output variable's existing scale factor, and may be any double precision number, including zero.

Returns:
0 if scale factor is reset successfully. An `index` out of range will return -1.

See also:
applyOutputScaleFactorByVarID
getOutputScaleFactorByIndex

2.4.2.2 int applyOutputScaleFactorByVarID (const char * *varID*, const double *factor*) `[inherited]`

This function should be used with extreme caution. The default scale factor for each output variable is unity. Each time this function is used, it multiplies by `factor` the current value of scale factor associated with the output variable whose *varID* matches the input `varID`.

Parameters:
varID is a short string without whitespace, such as "MACH02", which uniquely defines the output variable.

factor is the new scale factor to be applied multiplicatively to the output variable's existing scale factor, and may be any double precision number, including zero.

Returns:
0 if scale factor is reset successfully. If the `varID` input does not match any output variable ID within the DOM, the returned value is -1.

See also:
applyOutputScaleFactorByIndex
getOutputScaleFactorByVarID

2.4.2.3 int getNumberOfOutputs () `[inherited]`

The returned value counts all outputs found in the DOM (explicitly defined outputs, and results of function evaluations or MathML expression evaluations which do not also form calculation inputs). It may be used as the upper limit for an index to outputs.

Returns:
Total number of all output variables defined in the XML file, implicit and explict, and successfully loaded into the DOM.

2.4.2.4 double getOutputScaleFactorByIndex (const int *index*) `[inherited]`

The default scale factor for each output variable is unity. However, a reckless programmer can use applyOutputScaleFactorByIndex to change this value to any double precision number (including 0.0), so this function allows any such changes to be tracked, typically immediately prior to performing a computation of the output variable referenced by `index`.

Parameters:
index has a range from 0 to (getNumberOfOutputs() - 1), and selects the output variable to be addressed.

Returns:
The current accumulated scale factor for output variable `index` is returned, which may be any double precision number, including zero. An `index` out of range will return NaN.

2.4.2.5 double getOutputScaleFactorByVarID (const char * *varID*) [inherited]

The default scale factor for each output variable is unity. However, a reckless programmer can use applyOutputScaleFactorByVarID to change this value to any double precision number (including 0.0), so this function allows any such changes to be tracked, typically immediately prior to performing a computation of the output variable referenced by `varID`.

Parameters:

varID is a short string without whitespace, such as "MACH02", which uniquely defines the output variable.

Returns:

The current accumulated scale factor for the output variable whose *varID* matches the input `varID` will be returned. It may be any double precision number, including zero. If the `varID` input does not match any output variable ID within the DOM, the return value is NaN.

2.4.2.6 char * getOutputVariable (const int *index*, int) [inherited]

This allows a slightly strange extension of DAVE-ML usage. is use is not recommended unless you know what you are doing and are sure that you really need it. A gridded table of strings may be set up, and accessed in the same way as a tabular function. The array must be one-dimensionsional, and its breakpoints must be $0, 1, 2, \ldots n$ for (n-1) strings. Its input *variableDef* must have the same range as the breakpoints, and the output *variableDef* must be set to require 0th order polynomial interpolation. The strings can be delimited by any of: tab, newline, comma, semicolon. Do not start or end the strings with excess whitespace. Janus detects a string table by looking for non-numeric characters, so a table consisting entirely of numeric data will never be detected as a string. Note the string table can only be interrogated by output variable index, not by varID or any of the other indices.

Parameters:

index has a range from 0 to (getNumberOfOutputs() - 1), and selects the output variable to be addressed.

int second input is a dummy to allow the function to be overloaded.

Returns:

a string pointer to the output variable based on the current input state.

2.4.2.7 double getOutputVariable (const int *index*) [inherited]

This fulfils the basic purpose of the Janus class. It is used during run-time to evaluate output variables defined (explicitly or implicitly) within the XML source, based on independent variable values supplied to the instanced Janus class. The independent variable values must be set to required values, passing the aircraft state to the Janus instance using setIndependentVariableByIndex or the other value-input functions, before this function is used. One possible way of applying the Janus class to perform an output variable evaluation is:

```
int outputNumber = 0;
for ( int i = 0 ;
  i < prop.getNumberOfIndependentVariables( outputNumber ) ; i++ ) {
  cout << "\n  Enter value for "
       << prop.getIndependentVariableID( outputNumber, i )
       << " : ";
  double x;
```

```
  cin >> x;
  prop.setIndependentVariableByIndex( outputNumber, i, x );
}
double y = prop.getOutputVariable( outputNumber );
```

Parameters:

index has a range from 0 to (getNumberOfOutputs() - 1), and selects the output variable to be addressed.

Returns:

A double precision value containing the value of the output after all relevant computations based on the current input state. An `index` out of range will return NaN.

See also:

setIndependentVariableByIndex
setVariableByIndex
setVariableByID
getOutputVariableByVarID

2.4.2.8 double getOutputVariableByVarID (const char * *varID*) `[inherited]`

This is an alternative approach to the basic purpose of the Janus class. It is used during run-time to evaluate output variables defined (explicitly or implicitly) within the XML source, based on independent variable values supplied to the instanced Janus class. The independent variable values must be set to required values, passing the aircraft state to the Janus instance using setIndependentVariableByIndex or the other value-input functions, before this function is used.

Parameters:

varID is a short string without whitespace, such as "MACH02", which uniquely defines the output variable.

Returns:

A double precision value containing the value of the output variable whose *varID* matches the input `varID` after all relevant computations based on the current input state. If the input does not match any output variable ID within the DOM, the return value is NaN.

See also:

setIndependentVariableByIndex
setVariableByIndex
setVariableByID
getOutputVariable

2.4.2.9 char * getOutputVariableDescription (const int *index*) `[inherited]`

An output variable's *description* consists of a string of arbitrary length, which can include tabs and new lines as well as alphanumeric data. This means pretty formatting of the XML source will also appear in the description. Since description of a variable is optional, the returned string may be blank.

Parameters:

index has a range from 0 to (getNumberOfOutputs() - 1), and selects the output variable to be addressed.

Returns:

A pointer to an XML *variableDef* tag's *description* child element contents string is returned. An `index` out of range will return a zero pointer.

2.4.2.10 char ∗ getOutputVariableID (const int *index*) `[inherited]`

An output variable's *varID* attribute is normally a short string without whitespace, such as "MACH02", which uniquely defines the variable. It may be used for indexing. This function may be used by the calling program to determine the variable ID associated with each output variable location in the DOM.

Parameters:

index has a range from 0 to (getNumberOfOutputs() - 1), and selects the output variable to be addressed.

Returns:

A pointer to an XML *variableDef* tag's *varID* attribute string is returned. An `index` out of range will return a zero pointer.

2.4.2.11 int getOutputVariableIndex (const char ∗ *varID*) `[inherited]`

An output variable's *varID* attribute is uniquely related to the *variable* and may be used as an index. This function is used by the calling program to establish the order of output variables within the DOM, since it is always more efficient to address an output variable by numeric index than by variable ID.

Parameters:

varID is a short string without whitespace, such as "MACH02", which uniquely defines the output variable.

Returns:

An `index` in the range from 0 to (getNumberOfOutputs() - 1), corresponding to the output variable whose *varID* matches the supplied `varID`. If the input does not match any dependent variable ID within the DOM, the returned value is -1.

2.4.2.12 char ∗ getOutputVariableName (const int *index*) `[inherited]`

An output variable's *name* attribute is a string of arbitrary length, but normally short. It is not used for indexing, and therefore need not be unique (although uniqueness may aid both programmer and user), but should comply with the AIAA draft standard.

Parameters:

index has a range from 0 to (getNumberOfOutputs() - 1), and selects the output variable to be addressed.

Returns:

A pointer to an XML *variableDef* tag's *name* attribute string is returned. An `index` out of range will return a zero pointer.

2.4.2.13 char ∗ getOutputVariableUnits (const int *index*) [inherited]

An output variable's *units* attribute is a string of arbitrary length, but normally short, and complying with the format requirements chosen by Air Vehicles Division (AVD) Flight Systems Branch (FS) [Brian, 2004] in accordance with Systeme International d'Unites (SI) and other systems.

Parameters:

index has a range from 0 to (getNumberOfOutputs() - 1), and selects the output variable to be addressed.

Returns:

A pointer to an XML *variableDef* tag's *units* attribute string is returned. An `index` out of range will return a zero pointer.

2.5 Variables of All Types

Enumerations

- enum Janus::VariableType {
  Janus::FUNCTION, Janus::FUNCTION_INTERNAL, Janus::FUNCTION_OUTPUT, Janus::MATHML,
  Janus::MATHML_INTERNAL, Janus::MATHML_OUTPUT, Janus::ISOUTPUT, Janus::ISINPUT,
  Janus::ERRORVT }

Functions

- int Janus::getNumberOfVariables ()
- double Janus::getVariableByIndex (const int index)
- double Janus::getVariableByVarID (const char ∗varID)
- char ∗ Janus::getVariableDescription (const int index)
- char ∗ Janus::getVariableID (const int index)
- int Janus::getVariableIndex (const char ∗varID)
- char ∗ Janus::getVariableName (const int index)
- VariableType Janus::getVariableType (const int index)
- char ∗ Janus::getVariableUnits (const int index)
- int Janus::setVariableByID (const char ∗varID, const double x)
- int Janus::setVariableByIndex (const int index, const double x)

2.5.1 Detailed Description

All the *variableDef* functions use an `index` based on the *variableDef* elements at DTD Level 1 in the XML dataset. Each variable referenced by the `index` can be used by multiple functions, and can be dependent or independent. The order of variable definitions within the DOM is arbitrary and the calling program is responsible for determining which `index` to address. For example (also see Janus::getNumberOfVariables and Janus::getVariableID):

```
int nv = prop.getNumberOfVariables();
for ( int i = 0 ; i < nv ; i++ ) {
  cout << "  Variable " << i << " : \n"
  << "   ID           : "
  << prop.getVariableID( i ) << "\n";
}
```

The `index` also addresses a corresponding location in a static array of variable current values within the instance's data.

2.5.2 Enumeration Type Documentation

2.5.2.1 enum VariableType `[inherited]`

This enum is used within the class to determine the source of an output variable, and by calling programs to determine whether a variable is an input, or an output of various types. An array of these enums is established during instantiation, and is accessed through getVariableType.

Enumerator:

FUNCTION This *variableDef* is referenced as a dependent variable by a *function* definition, and its value will therefore be determined by a tabulated function evaluation. It will be available as a Janus output.

FUNCTION_INTERNAL This *variableDef* is referenced as a dependent variable by a *function* definition, and its value will therefore be determined by a tabulated function evaluation. It is also referenced as an independent variable by another calculation, and is not available as a Janus output.

FUNCTION_OUTPUT This *variableDef* is the result of a tabulated function evaluation, but is also used as an independent variable for another computation. It also has been explicitly defined as an output by its child node.

MATHML This *variableDef* includes a *calculation* child element, and its value will therefore be determined by by a MathML function evaluation. It will be available as a Janus output.

MATHML_INTERNAL This *variableDef* includes a *calculation* child element, and its value will therefore be determined by by a MathML function evaluation. It is also referenced as an independent variable by another calculation, and is not available as a Janus output.

MATHML_OUTPUT This *variableDef* is the result of a MathML computation, but is also used as an independent variable for another computation. It also has been explicitly defined as an output by its child node.

ISOUTPUT This *variableDef* is explicitly defined as an output by its child node, **and** is not the product of either a tabulated function or MathML evaluation.

ISINPUT This *variableDef* has none of the possible output attributes and should be treated as an input.

ERRORVT Used as a flag, indicates function or variable index out of range.

2.5.3 Function Documentation

2.5.3.1 int getNumberOfVariables () `[inherited]`

This procedure returns the total number of variables in the DOM. It includes all variables, makes no distinction between variable types, and provides no indication of whether they are dependent, independent, constant, output, or unused.

Returns:

Total number of all variables defined in the XML file and successfully loaded into the DOM.

2.5.3.2 double getVariableByIndex (const int *index*) `[inherited]`

This function provides a means of determining the current values of all variables defined within a Janus instance, whether independent or otherwise. Each of these values corresponds to a *variableDef*. For example:

```
int nv = prop.getNumberOfVariables();
for ( int i = 0 ; i < nv ; i++ ) {
  cout << "  Variable " << i << " : \n"
      << "   Value        : "
      << prop.getVariableByIndex( i ) << "\n";
}
```

Note that no function evaluations are performed by this procedure, so if any variable has been set by other means since the last function evaluation, dependent variable values may not correspond to their related independent variable values.

Parameters:

index has a range from 0 to (getNumberOfVariables() - 1), and selects the variable value to be addressed from the list of *VariableDef* s at DTD Level 1 of the DOM. It addresses the corresponding location in a static array within the instance's data structures.

Returns:

A double precision value containing the current variable value. An `index` out of range will return NaN.

See also:

setVariableByIndex
getVariableByVarID

2.5.3.3 double getVariableByVarID (const char ∗ *varID*) `[inherited]`

This function provides an alternative means of determining the current values of all variables defined within a Janus instance, whether independent or otherwise. Each of these values corresponds to a *variableDef*.

Note that no function evaluations are performed by this procedure, so if any variable has been set by other means since the last function evaluation, dependent variable values may not correspond to their independent variable values.

Parameters:

varID is a short string without whitespace, such as "MACH02", which uniquely defines the variable of which it is an attribute.

Returns:

A double precision value containing the current variable value. If the input does not match any dependent variable ID within the DOM, the returned value is NaN.

See also:

setVariableByVarID
getVariableByIndex

2.5.3.4 char ∗ getVariableDescription (const int *index*) `[inherited]`

A variable's *description* consists of a string of arbitrary length, which can include tabs and new lines as well as alphanumeric data. This means pretty formatting of the XML source will also appear in the description. Since description of a variable is optional, the returned string may be blank.

Parameters:

index has a range from 0 to (getNumberOfVariables() - 1), and selects the variable to be addressed from the list of *VariableDef* s at DTD Level 1 of the DOM.

Returns:

A pointer to an XML *variableDef* tag's *description* child element contents string is returned. An `index` out of range will return a zero pointer.

2.5.3.5 char * getVariableID (const int *index*) [inherited]

A variable's *varID* attribute is normally a short string without whitespace, such as "MACH02", which uniquely defines the variable. It may be used for indexing of all variable definitions, without distinction between variable types, and without requiring to know whether they are dependent, independent, constant, output, or unused. This function provides the means to determine the identities of the variables defined at sequential locations within the DOM.

Parameters:
index has a range from 0 to (getNumberOfVariables() - 1), and selects the variable to be addressed from the list of *VariableDef* s at DTD Level 1 of the DOM.

Returns:
A pointer to an XML *variableDef* tag's *varID* attribute string is returned. An `index` out of range will return a zero pointer.

2.5.3.6 int getVariableIndex (const char * *varID*) [inherited]

A variable's *varID* attribute is uniquely related to the *variableDef* and may be used as an index. This function is used by the calling program to establish the order of variable definitions, including all types within the DOM, since it is always more efficient to address a variable by numeric index than by variable ID. The returned integer value may be used to address all *variable'-related* attributes, child nodes or data elements.

Parameters:
varID is a short string without whitespace, such as "MACH02", which uniquely defines the variable of which it is an attribute.

Returns:
An `index` in the range from 0 to (getNumberOfVariables() - 1), corresponding to the variable whose *varID* matches the input `varID`. If the input does not match any dependent variable ID within the DOM, the returned value is -1.

2.5.3.7 char * getVariableName (const int *index*) [inherited]

A variable's *name* attribute is a string of arbitrary length, but normally short. It is not used for indexing, and therefore need not be unique (although uniqueness may aid both programmer and user), but should comply with the AIAA draft standard.

Parameters:
index has a range from 0 to (getNumberOfVariables() - 1), and selects the variable to be addressed from the list of *VariableDef* s at DTD Level 1 of the DOM.

Returns:
A pointer to an XML *variableDef* tag's *name* attribute string is returned. An `index` out of range will return a zero pointer.

2.5.3.8 Janus::VariableType getVariableType (const int *index*) [inherited]

A variable which is specified as an output, a function evaluation result, or a MathML function should not normally have its value set directly by the calling program. This function allows the caller to determine a variable's status in this regard.

Parameters:

index has a range from 0 to (getNumberOfVariables() - 1), and selects the variable to be addressed from the list of *VariableDef* s at DTD Level 1 of the DOM.

Returns:

The VariableType is returned on successful completion. If `index` is out of range this function will return ERRORVT.

2.5.3.9 char ∗ getVariableUnits (const int *index*) `[inherited]`

A variable's *units* attribute is a string of arbitrary length, but normally short, and complying with the format requirements chosen by AVD FS [Brian, 2004] in accordance with SI and other systems.

Parameters:

index has a range from 0 to (getNumberOfVariables() - 1), and selects the variable to be addressed from the list of *VariableDef* s at DTD Level 1 of the DOM.

Returns:

A pointer to an XML *variableDef* tag's *units* attribute string is returned. An `index` out of range will return a zero pointer.

2.5.3.10 int setVariableByID (const char ∗ *varID*, const double *x*) `[inherited]`

This function provides an alternative means to set the current values of any variables defined within a Janus instance, whether independent or otherwise. Each of these values corresponds to a *variableDef*. For example:

```
int retVal = MachCoeff.setVariableByID( "Mach", 0.95);
if ( 0 == retVal ) {
  cout << "\n Mach value set ... \n";
}
```

Note that this function allows all variables to be modified, potentially overwriting function outputs, without maintaining compatibility between input and output variables. In normal use, the calling program must ensure that it only addresses input variables.

Parameters:

varID is a short string without whitespace, such as "MACH02", which uniquely defines the variable of which it is an attribute.

x is the double precision value to which the current value of the indexed variable will be set.

Returns:

0 is returned on successful completion. If the input does not match any dependent variable ID within the DOM, the returned value is -1. Addressing an internal or output variable (highly undesirable in general) will return 1 on successful completion.

See also:

setVariableByIndex
getVariableByVarID

2.5.3.11 int setVariableByIndex (const int *index*, const double *x*) [inherited]

This function provides the means to set the current values of any variables defined within a Janus instance, whether independent or otherwise. Each of these values corresponds to a *variableDef*. For example, setting all input variables before evaluation:

```
char fileName[] = "pika_aero.xml";
Janus aeroCoeff( fileName );
int nv = aeroCoeff.getNumberOfVariables();
for ( int i = 0 ; i < nv ; i++ ) {
  if ( ISINPUT == aeroCoeff.getVariableType( i ) ) {
    cout << " Variable name : "
         << aeroCoeff.getVariableName( i )
         << "\n Enter value : ";
    double x;
    cin >> x;
    int result = aeroCoeff.setVariableByIndex( i, x );
    if ( 0 == result ) {
      cout << "\n Variable "
           << aeroCoeff.getVariableName( i ) << " set ...\n";
    }
  }
}
int nf = aeroCoeff.getNumberOfOutputs();
for ( int i = 0 ; i < nf ; i++ ) {
  double y = aeroCoeff.getOutputVariable( i );
  cout << "\n Function " << i << " value = " << y << "\n";
}
```

Note that this function allows a variable to be modified without reference to its possible use as an input variable for other output variables, or to its possible origin as an output from another function, and without maintaining compatibility between input and output variables.

Note that this function allows all variables to be modified, potentially overwriting function outputs, without maintaining compatibility between input and output variables. In normal use, the calling program must ensure that it only addresses input variables.

Parameters:

index has a range from 0 to (getNumberOfVariables() - 1), and selects the variable to be addressed from the list of *VariableDef* s at DTD Level 1 of the DOM. It addresses the corresponding location in a static array within the instance's data structures.

x is the double precision value to which the current value of the indexed variable will be set.

Returns:

0 is returned on successful completion. An `index` out of range will return -1. Addressing an internal or output variable (highly undesirable in general) will return 1 on successful completion.

See also:

setVariableByID
getVariableByIndex

2.6 Independent Variables

Enumerations

- enum Janus::Extrapolation {
  Janus::NEITHER, Janus::MINEX, Janus::MAXEX, Janus::BOTH,
  Janus::XMIN, Janus::XMAX, Janus::ERROREX }
- enum Janus::Interpolation {
  Janus::LINEAR, Janus::POLY, Janus::CSPLINE, Janus::LEGENDRE,
  Janus::ERRORIN }

Functions

- double Janus::getIndependentVariableByIndex (const int indexf, const int indexv)
- double Janus::getIndependentVariableDataMax (const int indexf, const int indexv)
- double Janus::getIndependentVariableDataMin (const int indexf, const int indexv)
- double Janus::getIndependentVariableDataStep (const int indexf, const int indexv)
- char ∗ Janus::getIndependentVariableDescription (const int indexf, const int indexv)
- Extrapolation Janus::getIndependentVariableExtrapolation (const int indexf, const int indexv)
- Extrapolation Janus::getIndependentVariableExtrapolationFlag (const int indexf, const int indexv)
- char ∗ Janus::getIndependentVariableID (const int indexf, const int indexv)
- int Janus::getIndependentVariableIndex (const int indexf, const char ∗varID)
- Interpolation Janus::getIndependentVariableInterpolation (const int indexf, const int indexv)
- double Janus::getIndependentVariableMax (const int indexf, const int indexv)
- double Janus::getIndependentVariableMin (const int indexf, const int indexv)
- char ∗ Janus::getIndependentVariableName (const int indexf, const int indexv)
- int Janus::getIndependentVariableOrder (const int indexf, const int indexv)
- VariableType Janus::getIndependentVariableType (const int indexf, const int indexv)
- char ∗ Janus::getIndependentVariableUnits (const int indexf, const int indexv)
- int Janus::getNumberOfIndependentVariables (const int index)
- int Janus::setIndependentVariableByIndex (const int indexf, const int indexv, const double x)

2.6.1 Detailed Description

Independent variables are those which form the inputs to computation of an output variable value, and are defined relative to the output variable which requires them. Therefore independent variables are always referenced by Janus through output variables. The order of the *variableDefs* defining output variables at DTD Level 1 within the DOM is arbitrary and the calling program is responsible for determining which output variable to address. For output variables dependent on more than one input variable, the order of independent variable references by the output variable is also arbitrary and must be determined by the calling program. Note that a variable considered independent by one output may itself be the output of another computation, and one variable may be used to compute many outputs.

The independent variable procedures use a first index based on the output variable, and a second index based on the list of input variables required by it. The input variable, although it may be used by multiple outputs, is thus referenced through a different index for each output variable.

2.6.2 Enumeration Type Documentation

2.6.2.1 enum Extrapolation `[inherited]`

The optional *extrapolate* attribute of *independentVarRef* and *independentVarPts* child nodes of a *function* in the XML dataset governs the treatment of the function's independent variables when their requested input value exceeds the data range available. Each input variable has a limited data range, which is determined by the extremities of the list of points for single-variable, directly-defined functions, and by the extremities of the breakpoints for functions defined by reference. Note that the same input variable can have different extrapolation treatments for different functions.

Variable references for functions defined by reference can also specify minimum and maximum values, which do not necessarily coincide with the extremities of the breakpoints. The *extrapolate* attribute does not allow for exceedance of any defined minimum and maxmimum values.

This enum can take its value from the *extrapolate* attribute, and may be used to determine any extrapolation allowable for each input variable used in each function. Its normal values in this usage are NEITHER, MINEX, MAXEX, or BOTH (see getIndependentVariableExtrapolation). It may also indicate activation of data range and extrapolation constraints during a Janus function evaluation. Its normal values in this usage are NEITHER, MINEX, MAXEX, XMIN, or XMAX (see getIndependentVariableExtrapolationFlag, getIndependentVariableMin, getIndependentVariableMax).

Enumerator:

NEITHER No extrapolation allowed (When used as a flag, indicates no extrapolation was required during computation).

MINEX Extrapolation below data range minimum allowed (When used as a flag, indicates specified minimum value constraint was activated during computation).

MAXEX Extrapolation above data range maximum allowed (When used as a flag, indicates specified maximum value constraint was activated during computation).

BOTH Extrapolation above or below data range limits allowed.

XMIN Used as a flag, indicates input value was below data range minimum.

XMAX Used as a flag, indicates input value was above data range maximum

ERROREX Used as a flag, indicates function or variable index out of range (see getIndependentVariableExtrapolation and getIndependentVariableExtrapolationFlag).

2.6.2.2 enum Interpolation `[inherited]`

The optional *interpolationType* attribute of *independentVarRef* and *independentVarPts* child nodes of a *function* in the XML dataset governs the form of interpolation to be used in that variable's degree of freedom when evaluating the *function* between gridded data points. Note that the same input variable can have different interpolation treatments in different functions.

This enum can take its value from the *interpolationType* attribute, and may be used to determine the required form of interpolation for each input variable used in each function. Its normal values in this usage are LINEAR or POLY (see getIndependentVariableInterpolation). Other interpolation types are not yet implemented. The enum is also used within the Janus instance for other purposes.

Enumerator:

LINEAR Interpolation in this degree of freedom is linear, maintaining continuity of data, but with derivatives discontinuous across breakpoints, unless *interpolationOrder* is set to 0 for this degree of freedom.

POLY Interpolation in this degree of freedom is polynomial, of order specified by *interpolationOrder*, maintaining continuity of data. Derivatives are continuous if the number of breakpoints matches *interpolationOrder* + 1, not otherwise.

CSPLINE Not in use at present.

LEGENDRE Not in use at present.

ERRORIN Used as a flag, indicates function or variable index out of range (see getIndependentVariableInterpolation).

2.6.3 Function Documentation

2.6.3.1 double getIndependentVariableByIndex (const int *indexf*, const int *indexv*) [inherited]

This function provides a means of determining the current values of all variables defined within a Janus instance which are required as input signals for an output signal evaluation. For example:

```
char fileName[] = "pika_aero.xml";
Janus aeroCoeff( fileName );
int nf = aeroCoeff.getNumberOfOutputs();
for ( int i = 0 ; i < nf ; i++ ) {
  cout << "  Output " << i << " : \n"
       << "   Name                     : "
       << aeroCoeff.getOutputName( i ) << "\n";
  iv = aeroCoeff.getNumberOfIndependentVariables( i );
  for ( int j = 0 ; j < iv ; j++ ) {
    cout << " Independent variable name : "
         << aeroCoeff.getIndependentVariableName( i, j )
         << "\n                     value : "
         << aeroCoeff.getIndependentVariableByIndex( i, j )
         << "\n";
  }
}
```

Note that no *function* or MathML expression evaluations are performed by this procedure. It merely makes the values of independent variables, set by other procedures, accessible to the calling program.

Parameters:

indexf has a range from 0 to (getNumberOfOutputs() - 1), and selects the output variable to be addressed.

indexv has a range from 0 to (getNumberOfIndependentVariables (indexf) - 1), and selects the independent variable to be addressed through the output variable which uses it.

Returns:

A double precision value containing the current value value of the independent variable selected, for immediate use to compute the output variable selected. If the independent variable is undefined or inapplicable, or an index is out of range, either `indexf` for the output variable or `indexv` for its independent variables, this function will return a NaN.

2.6.3.2 double getIndependentVariableDataMax (const int *indexf*, const int *indexv*) [inherited]

The maximum value of an independent variable used in a computation (see getIndependentVariableMax) is not necessarily the same as the maximum value of data supplied for the variable. This function provides the calling program access to the actual input data limit. For gridded data, it returns the highest-valued end breakpoint in the selected degree of freedom. For ungridded data, it returns the highest-value in the selected degree of freedom which attaches to any data point. This function is not applicable to MathML-based computations at present.

Parameters:

indexf has a range from 0 to (getNumberOfOutputs() - 1), and selects the output variable to be addressed.

indexv has a range from 0 to (getNumberOfIndependentVariables (indexf) - 1), and selects the independent variable to be addressed through the output variable which uses it.

Returns:

A double precision value containing the maximum supplied data value of the independent variable selected. If the output variable depends on a MathML computation, or an index is out of range, either `indexf` for the output variable or `indexv` for its independent variables, this function will return a NaN.

2.6.3.3 double getIndependentVariableDataMin (const int *indexf*, const int *indexv*) `[inherited]`

The minimum value of an independent variable used in a computation (see getIndependentVariableMin) is not necessarily the same as the minimum value of data supplied for the variable. This function provides the calling program access to the actual input data limit. For gridded data, it returns the lowest-valued end breakpoint in the selected degree of freedom. For ungridded data, it returns the lowest-value in the selected degree of freedom which attaches to any data point. This function is not applicable to MathML-based computations at present.

Parameters:

indexf has a range from 0 to (getNumberOfOutputs() - 1), and selects the output variable to be addressed.

indexv has a range from 0 to (getNumberOfIndependentVariables (indexf) - 1), and selects the independent variable to be addressed through the output variable which uses it.

Returns:

A double precision value containing the minimum supplied data value of the independent variable selected. If the output variable depends on a MathML computation, or an index is out of range, either `indexf` for the output variable or `indexv` for its independent variables, this function will return a NaN.

2.6.3.4 double getIndependentVariableDataStep (const int *indexf*, const int *indexv*) `[inherited]`

Where outputs are based on tabular data, some indication of the density or granularity of that data may be useful to the calling program. This function computes an increment, for each input degree of freedom, which may be used to step across the input data range while capturing the detail contained within the data. For uniform gridded data, it returns the difference between successive breakpoint values. For non-uniform gridded data, it returns a value 1 average deviation less than the average non-zero difference between successive breakpoint values. For ungridded data, it returns a value 1 average deviation below the average non-zero dimension of the Delaunay tesselations in the requested degree of freedom. This function is not applicable to MathML-based computations at present.

Parameters:

indexf has a range from 0 to (getNumberOfOutputs() - 1), and selects the output variable to be addressed.

indexv has a range from 0 to (getNumberOfIndependentVariables (indexf) - 1), and selects the independent variable to be addressed through the output variable which uses it.

Returns:

A double precision value containing the suggested step increment value for the independent variable selected. If the output variable depends on a MathML computation, or an index is out of range, either `indexf` for the output variable or `indexv` for its independent variables, this function will return a NaN.

2.6.3.5 char * getIndependentVariableDescription (const int *indexf*, const int *indexv*) `[inherited]`

An independent variable's *description* consists of a string of arbitrary length, which can include tabs and new lines as well as alphanumeric data. This means pretty formatting of the XML source will also appear in the description. Since description of a variable is optional, the returned string may be blank.

Parameters:

indexf has a range from 0 to (getNumberOfOutputs() - 1), and selects the output variable to be addressed.

indexv has a range from 0 to (getNumberOfIndependentVariables (indexf) - 1), and selects the independent variable to be addressed through the output variable which uses it.

Returns:

A pointer to an XML *variableDef* tag's *description* child element contents string is returned. An index out of range, either `indexf` for the output variable or `indexv` for its independent variables, will return a zero pointer.

2.6.3.6 Janus::Extrapolation getIndependentVariableExtrapolation (const int *indexf*, const int *indexv*) `[inherited]`

The *extrapolate* attribute of an independent variable referenced by an output variable describes any allowable extrapolation in the independent variable's degree of freedom contributing to computation of the output. freedom. This function makes that characteristic available to the calling program, from an Extrapolation enum within the Janus instance.

The *extrapolate* attribute is only applicable to an output variable defined in terms of a tabular *function* (i.e. constants never require extrapolation, and MathML computations can incorporate extrapolation within their defining computations). The *extrapolate* attribute is optional for all degrees of freedom for any *function* within the XML dataset, and if it is not set for any particular degree of freedom then the enum representing its value within the Janus instance defaults to NEITHER.

When the returned value is NEITHER, MINEX, or MAXEX, constraining the independent variable at neither end, the maximum, or the minimum respectively, the constrained independent variable value used for the *function* evaluation will be the more limiting of:

Min Constraints	Max Constraints
lowest independentVarPts *or* lowest breakpoint	highest independentVarPts *or* highest breakpoint
min attribute	*max* attribute

Parameters:

indexf has a range from 0 to (getNumberOfOutputs() - 1), and selects the output variable to be addressed.

indexv has a range from 0 to (getNumberOfIndependentVariables (indexf) - 1), and selects the independent variable to be addressed through the output variable which uses it.

Returns:

An Extrapolation enum containing the extrapolation constraint on the independent variable selected, when used to compute the output variable selected. If the *extrapolate* attribute is inapplicable, or an index is out of range, either `indexf` for the output variable or `indexv` for its independent variables, this function will return ERROREX.

2.6.3.7 Janus::Extrapolation getIndependentVariableExtrapolationFlag (const int *indexf*, const int *indexv*) `[inherited]`

The extrapolation flag is only applicable to an output variable defined in terms of a tabular *function* (i.e. constants never require extrapolation, and MathML computations can incorporate within their defining computations).

For each independent degree of freedom for each *function* defined within the DOM, the Janus instance maintains a flag in the form of an Extrapolation enum which indicates whether the most recent evaluation of the selected *function* activated any extrapolation constraint on the selected independent variable. This constraint may relate to a *min* or *max* attribute, or to the extremities of *independentVarPts* or to the extremities of breakpoints. The possible constraints are tabulated under getIndependentVariableMin and getIndependentVariableMax.

Parameters:

indexf has a range from 0 to (getNumberOfOutputs() - 1), and selects the output variable to be addressed.

indexv has a range from 0 to (getNumberOfIndependentVariables (indexf) - 1), and selects the independent variable to be addressed through the output variable which uses it.

Returns:

An Extrapolation enum indicating the most significant activated data range limit on the independent variable selected, during the last evaluation of the output variable selected. If the *extrapolate* attribute is inapplicable, or an index is out of range, either `indexf` for the output variable or `indexv` for its independent variables, this function will return ERROREX.

2.6.3.8 char * getIndependentVariableID (const int *indexf*, const int *indexv*) `[inherited]`

An independent variable's *varID* attribute is normally a short string without whitespace, such as "MACH02", which uniquely defines the variable. It may be used for indexing. This function may be used by the calling program to determine the input variable ID associated with each independent variable required by each output varaible defined in the DOM.

Parameters:

indexf has a range from 0 to (getNumberOfOutputs() - 1), and selects the output variable to be addressed.

indexv has a range from 0 to (getNumberOfIndependentVariables (indexf) - 1), and selects the independent variable to be addressed through the output variable which uses it.

Returns:

A pointer to an XML *variableDef* tag's *varID* attribute string is returned. An index out of range, either `indexf` for the output variable or `indexv` for its independent variables, will return a zero pointer.

2.6.3.9 int getIndependentVariableIndex (const int *indexf*, const char ∗ *varID*) `[inherited]`

When an independent variable is referred to through an output signal which uses the variable, the variable's *varID* attribute is uniquely related to the output signal and may be used as an index. This function is used by the calling program to establish the order of independent variable references through the output variable, since it is always more efficient to address a variable by numeric index than by variable ID.

Parameters:

indexf has a range from 0 to (getNumberOfOutputs() - 1), and selects the output variable to be addressed.

varID is a short string without whitespace, such as "MACH02", which uniquely defines the independent variable from the list associated with the output variable which uses it.

Returns:

An integer `index` in the range from 0 to (getNumberOfIndependentVariables (indexf) - 1), corresponding to the independent variable whose *varID* matches the input `varID`. If the input does not match any variable ID within the DOM, or the output variable `indexf` is out of range, returned value is -1.

2.6.3.10 Janus::Interpolation getIndependentVariableInterpolation (const int *indexf*, const int *indexv*) `[inherited]`

Interpolation type and order for gridded data are proposed additions to DAVE-ML. As of DAVE-ML Version 1.7b1 they had not been added to the official DTD. To use them until this addition takes place, ensure that the DTD version has a *dmn* suffix.

The *interpolationType* attribute of an independent variable referenced by an output variable describes the form of interpolation to be used in that independent variable's degree of freedom when computing the output variable's value based on interpolation between gridded data points. This function makes that information available to the calling program, from an Interpolation enum within the Janus instance.

The *interpolationType* attribute is only applicable to an output variable defined in terms of a tabular *function* (i.e. constants never require interpolation, and MathML computations can incorporate interpolation within their defining computations).

The *interpolationType* attribute is optional for all degrees of freedom for any *function* within the XML dataset, and if it is not set for any particular degree of freedom then the enum representing its value within the Janus instance defaults to LINEAR, which is identical to POLY of order 1 (order is specified by the *interpolationOrder* attribute and available to the calling program using getIndependentVariableOrder).

Applications are free to ignore this attribute (e.g. to sacrifice accuracy for speed in real time computations). The current Janus implementation limits POLY to order 3, and does not allow CSPLINE or LEGENDRE interpolations. Setting the order to 0 for either a linear or polynomial interpolation will limit values in that degree of freedom to discrete steps centred on the breakpoint values.

Parameters:

indexf has a range from 0 to (getNumberOfOutputs() - 1), and selects the output variable to be addressed.

indexv has a range from 0 to (getNumberOfIndependentVariables (indexf) - 1), and selects the independent variable to be addressed through the output variable which uses it.

Returns:

An Interpolation enum containing the form of interpolation for the independent variable selected, when used to compute the output variable selected. If the *interpolationType* attribute is inapplicable, or an index is out of range, either `indexf` for the output variable or `indexv` for its independent variables, this function will return ERRORIN.

2.6.3.11 double getIndependentVariableMax (const int *indexf*, const int *indexv*) `[inherited]`

The *max* attribute of an independent variable referenced by an output variable describes the allowable upper limit in the independent variable's degree of freedom contributing to computation of the output. This function makes that limit available to the calling program.

The *max* attribute is only applicable to an output variable defined in terms of a tabular *function* (i.e. constants never require a maximum to be specified, and MathML computations can incorporate limits within their defining computations). The *max* attribute is optional for all degrees of freedom for any function within the XML dataset, and if it is not set for any particular degree of freedom then the data may be extrapolated upwards without limit in that degree of freedom unless the *extrapolate* attribute indicates otherwise.

Note that a variable may be an independent input for multiple output variables, and may have a different *max* for each such output which is defined in terms of a *function*. Also, the *max* need not coincide with the maximum *independentVarPts* or breakpoint (x_{max}) for its degree of freedom.

The value (x) of an independent variable used for evaluation of a function is never greater than *max*, no matter what the input value is or what other constraints are applied. Within this constraint, the *max* attribute interacts with both the highest available value for its variable and the variable's *extrapolate* attribute (see getIndependentVariableExtrapolation and getIndependentVariableExtrapolationFlag), to define the input value used in a function evaluation. Whenever a constraint is activated during a function evaluation, the extrapolation flag is changed. The various possible combinations of constraining attributes and data limits are:

extrapolate attribute	x relative values	x used in computation	extrapolation flag after computation
any value	$x < max < x_{max}$	x	NEITHER
	$x < x_{max} < max$	x	NEITHER
neither / min	$max < x < x_{max}$	*max*	MAXEX
	$max < x_{max} < x$	*max*	MAXEX
	$x_{max} < x < max$	x_{max}	XMAX
	$x_{max} < max < x$	x_{max}	XMAX
max / both	$max < x < x_{max}$	*max*	MAXEX
	$max < x_{max} < x$	*max*	MAXEX
	$x_{max} < x < max$	x	XMAX
	$x_{max} < max < x$	*max*	MAXEX

Parameters:

indexf has a range from 0 to (getNumberOfOutputs() - 1), and selects the output variable to be addressed.

indexv has a range from 0 to (getNumberOfIndependentVariables (indexf) - 1), and selects the independent variable to be addressed through the output variable which uses it.

Returns:

A double precision value containing the maximum allowable value of the independent variable selected, when used to compute the output variable selected. If the *max* attribute is undefined or inapplicable, or an index is out of range, either `indexf` for the output variable or `indexv` for its independent variables, this function will return a NaN.

See also:

getIndependentVariableDataMax
getIndependentVariableExtrapolation
getIndependentVariableExtrapolationFlag

2.6.3.12 double getIndependentVariableMin (const int *indexf*, const int *indexv*) `[inherited]`

The *min* attribute of an independent variable referenced by an output variable describes the allowable lower limit in the independent variable's degree of freedom contributing to computation of the output. This function makes that limit available to the calling program.

The *min* attribute is only applicable to an output variable defined in terms of a tabular *function* (i.e. constants never require a minimum to be specified, and MathML computations can incorporate limits within their defining computations). The *min* attribute is optional for all degrees of freedom for *functions* within the XML dataset, and if it is not set for any particular degree of freedom then the data may be extrapolated downwards without limit in that degree of freedom unless the *extrapolate* attribute indicates otherwise.

Note that a variable may be an independent input for multiple output variables, and may have a different *min* for each such output which is defined in terms of a *function*. Also, the *min* need not coincide with the minimum *independentVarPts* or breakpoint (x_{min}) for its degree of freedom.

The value (x) of an independent variable used for evaluation of a function is never less than *min*, no matter what the input value is or what other constraints are applied. Within this constraint, the *min* attribute interacts with both the lowest available value for its variable and the variable's *extrapolate* attribute (see getIndependentVariableExtrapolation and getIndependentVariableExtrapolationFlag), to define the input value used in a function evaluation. Whenever a constraint is activated during a function evaluation, the extrapolation flag is changed. The various possible combinations of constraining attributes and data limits are:

extrapolate attribute	x relative values	x used in computation	extrapolation flag after computation
any value	$x_{min} < min < x$	x	NEITHER
	$min < x_{min} < x$	x	NEITHER
neither / max	$x_{min} < x < min$	*min*	MINEX
	$x < x_{min} < min$	*min*	MINEX
	$min < x < x_{min}$	x_{min}	XMIN
	$x < min < x_{min}$	x_{min}	XMIN
min / both	$x_{min} < x < min$	*min*	MINEX
	$x < x_{min} < min$	*min*	MINEX
	$min < x < x_{min}$	x	XMIN
	$x < min < x_{min}$	*min*	MINEX

Parameters:

indexf has a range from 0 to (getNumberOfOutputs() - 1), and selects the output variable to be addressed.

indexv has a range from 0 to (getNumberOfIndependentVariables (indexf) - 1), and selects the independent variable to be addressed through the output variable which uses it.

Returns:

A double precision value containing the minimum allowable value of the independent variable selected, when used to compute the output variable selected. If the *min* attribute is undefined or inapplicable, or an index is out of range, either `indexf` for the output variable or `indexv` for its independent variables, this function will return a NaN.

See also:

getIndependentVariableDataMin
getIndependentVariableExtrapolation
getIndependentVariableExtrapolationFlag

2.6.3.13 char * getIndependentVariableName (const int *indexf*, const int *indexv*) `[inherited]`

An independent variable's *name* attribute is a string of arbitrary length, but normally short. It is not used for indexing, and therefore need not be unique (although uniqueness may aid both programmer and user), but should comply with the AIAA draft standard.

Parameters:

indexf has a range from 0 to (getNumberOfOutputs() - 1), and selects the output variable to be addressed.

indexv has a range from 0 to (getNumberOfIndependentVariables (indexf) - 1), and selects the independent variable to be addressed through the output variable which uses it.

Returns:

A pointer to an XML *variableDef* tag's *name* attribute string is returned. An index out of range, either `indexf` for the output variable or `indexv` for its independent variables, will return a zero pointer.

2.6.3.14 int getIndependentVariableOrder (const int *indexf*, const int *indexv*) `[inherited]`

The *interpolationOrder* attribute of an independent variable referenced by an output variable describes the order of interpolation to be used in that variable's degree of freedom when when computing the output variable's value based on interpolation between gridded data points. This function makes that order available to the calling program, from static data within the Janus instance which is initialised with *interpolationOrder* data from the XML dataset.

The *interpolationOrder* attribute is only applicable to an output variable defined in terms of a tabular *function* (i.e. constants never require interpolation, and MathML computations can incorporate interpolation within their defining computations).

The *interpolationOrder* attribute is optional for all degrees of freedom for any *function* within the XML dataset, and if it is not set for any particular degree of freedom then its representation within the Janus instance defaults to 1, representing linear interpolation under either linear or polynomial interpolation types (type is specified by the *interpolationType* attribute and available to the calling program using getIndependentVariableInterpolation).

Applications are free to ignore this attribute (e.g. to sacrifice accuracy for speed in real time computations). The current Janus implementation allows linear interpolation of order 0 or 1 and polynomial interpolation of order 0 to 3 inclusive. Setting the order to 0 for either a linear or polynomial interpolation will limit values in that degree of freedom to discrete steps centred on the breakpoint values.

Parameters:

indexf has a range from 0 to (getNumberOfOutputs() - 1), and selects the output variable to be addressed.

indexv has a range from 0 to (getNumberOfIndependentVariables (indexf) - 1), and selects the independent variable to be addressed through the output variable which uses it.

Returns:

The interpolation order is returned on successful completion. If the *interpolationType* attribute is inapplicable, or an index is out of range, either `indexf` for the output variable or `indexv` for its independent variables, this function will return -1.

2.6.3.15 Janus::VariableType getIndependentVariableType (const int *indexf*, const int *indexv*) `[inherited]`

An independent variable which is also specified as an output, a function evaluation result, or a MathML function should not normally have its value set directly by the calling program. This function allows the caller to determine a variable's status in this regard.

Parameters:

indexf has a range from 0 to (getNumberOfOutputs() - 1), and selects the output variable to be addressed.

indexv has a range from 0 to (getNumberOfIndependentVariables (indexf) - 1), and selects the independent variable to be addressed through the output variable which uses it.

Returns:

The VariableType is returned on successful completion. If an index is out of range, either `indexf` for the output variable or `indexv` for its independent variables, this function will return ERRORVT.

2.6.3.16 char * getIndependentVariableUnits (const int *indexf*, const int *indexv*) [inherited]

An independent variable's *units* attribute is a string of arbitrary length, but normally short, and complying with the format requirements chosen by AVD FS [Brian, 2004] in accordance with SI and other systems.

Parameters:

indexf has a range from 0 to (getNumberOfOutputs() - 1), and selects the output variable to be addressed.

indexv has a range from 0 to (getNumberOfIndependentVariables (indexf) - 1), and selects the independent variable to be addressed through the output variable which uses it.

Returns:

A pointer to an XML *variableDef* tag's *units* attribute string is returned. An index out of range, either `indexf` for the output variable or `indexv` for its independent variables, will return a zero pointer.

2.6.3.17 int getNumberOfIndependentVariables (const int *index*) [inherited]

This procedure returns the number of independent variables associated with any output variable. Note an independent variable can be of any of the types in the enum VariableType.

Parameters:

index has a range from 0 to (getNumberOfOutputs() - 1), and selects the output variable to be addressed.

Returns:

Total number of independent variables referenced by the output variable selected. An `index` out of range will return -1.

2.6.3.18 int setIndependentVariableByIndex (const int *indexf*, const int *indexv*, const double *x*) [inherited]

This procedure is one of the means of setting an input variable value in a static array within the Janus instance's data structure, which corresponds to a *variableDef* referred to by a *function* to be evaluated.

Parameters:

indexf has a range from 0 to (getNumberOfOutputs() - 1), and selects the output variable to be addressed.

indexv has a range from 0 to (getNumberOfIndependentVariables (indexf) - 1), and selects the independent variable to be addressed through the output variable which uses it.

x is the double precision value to which the current value of the indexed variable will be set.

Returns:

0 is returned on successful completion. An index out of range, either `indexf` for the output variable or `indexv` for its independent variables, will return -1.

See also:

getIndependentVariableType

2.7 MathML Operations

Enumerations

- enum Janus::MathOp {
  Janus::PLUS, Janus::MINUS, Janus::TIMES, Janus::DIVIDE,
  Janus::POWER, Janus::SIN, Janus::COS, Janus::TAN,
  Janus::CSC, Janus::SEC, Janus::COT, Janus::ARCSIN,
  Janus::ARCCOS, Janus::ARCTAN, Janus::EXP, Janus::LN,
  Janus::LOG, Janus::ABS, Janus::EQUAL, Janus::NEQ,
  Janus::GT, Janus::LT, Janus::GEQ, Janus::LEQ,
  Janus::MIN, Janus::MAX, Janus::PIECEWISE, Janus::AND,
  Janus::OR, Janus::XOR, Janus::NOT, Janus::FACTORIAL,
  Janus::QUOTIENT, Janus::REM, Janus::FLOOR, Janus::CEILING,
  Janus::PI, Janus::NOOP }

2.7.1 Detailed Description

The type of function used to represent a particular data item is programmatically irrelevant to the modeller whose code uses a Janus instance to determine data values. However, the dataset developer must decide both the type of data (gridded, ungridded, or MathML) and other details of its representation. In the case of MathML data, this includes the available operators, since Janus only implements a subset of the many hundreds of MathML operators defined. For this purpose, the following enum shows all operators currently implemented within the Janus code.

These operators are implemented for real variables only, a subset of the MathML variable types which include rational, complex, integer and vector forms. The operators have all the normal range limits applicable to any mathematical operation.

2.7.2 Enumeration Type Documentation

2.7.2.1 enum MathOp `[inherited]`

This enum is used within the class to determine the type of mathematical operation to apply to one or more numeric arguments. Operations are unary (take only one argument), binary (take exactly two arguments), or n-ary (take one or more arguments). At present Janus arbitrarily limits each operation to a maximum of 64 arguments. A few operations can fall in more than one category (e.g. MINUS can be unary sign change or binary difference). In these cases Janus decides which operation is appropriate by counting the number of arguments.

Types are enumerated here, and where used in the programs, in order of expected decreasing frequency of use. This order can have a marginal effect on speed of execution.

Enumerator:

PLUS Sum (n-ary, n >= 1)

MINUS Difference (binary, second argument subtracted from first).

TIMES Product (n-ary, n >= 1)

DIVIDE Ratio (binary, first argument divided by second).

POWER Exponentiation (binary, first argument raised to power of second argument).

SIN Sine of input in radians (unary)

COS Cosine of input in radians (unary)

TAN Tangent of input in radians (unary)

CSC Cosecant, 1/sine of input in radians (unary)

SEC Secant, 1/cosine of input in radians (unary)

COT Cotangent, 1/tangent of input in radians (unary)

ARCSIN Computes the angle (in radians) whose sine is equal to the input, which has a range from -1 to 1 (unary).

ARCCOS Computes the angle (in radians) whose cosine is equal to the input, which has a range from -1 to 1 (unary).

ARCTAN Computes the angle (in radians) whose tangent is equal to the input, whose range is unbounded (unary).

EXP Computes the base of natural logarithms raised to the power of the unary input. This element represents the exponentiation function as described in Abramowitz and Stegun, section 4.2.

LN This element represents the `ln` function (natural logarithm) as described in Abramowitz and Stegun, section 4.1. It is unary.

LOG This element represents the `log` function. It is defined in Abramowitz and Stegun, Handbook of Mathematical Functions, section 4.1. If its first argument is a `logbase` element, it specifies the base and the second argument is the argument to which the function is applied using that base. If no `logbase` element is present, the base is assumed to be 10 and the function is unary.

ABS A unary operator which represents the absolute value of its argument. Often referred to as the modulus.

EQUAL This n-ary function is used to indicate that two or more quantities are equal. There must be at least two arguments.

NEQ This binary function represents the relation "not equal to" which returns true unless the two arguments are equal.

GT This n-ary function represents the relation "greater than" which returns true if each argument in turn is greater than the one following it. There must be at least two arguments.

LT This n-ary function represents the relation "less than" which returns true if each argument in turn is less than the one following it. There must be at least two arguments.

GEQ This element represents the n-ary "greater than or equal to" function, which returns true if each argument in turn is greater than or equal to the one following it. There must be at least two arguments.

LEQ This n-ary function represents the relation "less than or equal to" which returns true if each argument in turn is less or equal to the one following it. There must be at least two arguments.

MIN This is the n-ary operator used to represent the minimum of a set of elements. The elements may be listed explicitly or they may be described by a condition (condition not yet implemented in Janus). The elements must all be comparable if the result is to be well defined.

MAX This is the n-ary operator used to represent the maximum of a set of elements. The elements may be listed explicitly or they may be described by a domain of application (condition not yet implemented in Janus.

PIECEWISE The 'piecewise', 'piece', and 'otherwise' elements are used to support 'piecewise' declarations of the form $H(x) = 0$ if x less than 0 , $H(x) = 1$ otherwise. The 'piece' and 'otherwise' elements describe evaluation rules. If no rule applies or if more than one rule applies but they give different answers then the expression is undefined. This allows a function to be segmented.

AND This is the n-ary logical "and" operator. It is used to construct the logical expression which were it to be evaluated would have a value of "true" when all of its operands have a truth value of "true", and "false" otherwise.

OR The is the n-ary logical "or" operator. The constructed expression has a truth value of 'true' if at least one of its arguments is true.

XOR The is the n-ary logical "xor" operator. The constructed expression has a truth value of 'true' if an odd number of its arguments are true.

NOT This is the unary logical "not" operator. It negates the truth value of its single argument. e.g., not P is true when P is false and false when P is true.

FACTORIAL This is the unary operator used to construct factorials of positive integers, defined by $n! = n \times (n-1) \times \ldots \times 1$

QUOTIENT Quotient is the binary function used to represent the operation of integer division. Quotient(a,b) denotes q such that $a = b \times q + r$, with $|r|$ less than $|b|$ and $a \times r$ non-negative.

REM This is the binary operator used to represent the integer remainder $a \mod b$. For arguments a and b , such that $a = b \times q + r$ with $|r| < |b|$ it represents the value r .

FLOOR The floor element is used to denote the round-down (towards -infinity) operator.

CEILING The ceiling element is used to denote the round-up (towards +infinity) operator.

PI 3.14159265358979 is used in Janus to provide a reasonable double precision approximation

NOOP No operation, used to signal error condition

2.8 XML File Encryption or Decryption

Enumerations

- enum RsaKeyType { RSA_PRIVATE_KEY, RSA_PUBLIC_KEY }
- enum Janus::RsaKeyType { Janus::RSA_PRIVATE_KEY, Janus::RSA_PUBLIC_KEY }

Functions

- int createRsaKeys (void)
- int Janus::createRsaKeys (void)
- int setRsaKeyFileName (const char *fileName, const RsaKeyType keyType)
- int Janus::setRsaKeyFileName (const char *fileName, const RsaKeyType keyType)
- int writeEncryptedXmlFile (const char *fileName)
- int Janus::writeEncryptedXmlFile (const char *fileName)

2.8.1 Detailed Description

The Janus library can use the Apache XML-Security-C library and OpenSSL to allow XML aircraft datasets to be encrypted or decrypted. The encryption/decryption is performed within the DOM, and is transparent to the user provided the required security keys are available. Because decryption is performed at load time, as soon as the DOM is constructed in memory, there is no run-time performance penalty associated with using encrypted XML files. When compiling the Janus library, the encryption/decryption functionality is only included if the define -DENCRYPT_LIBS is used.

2.8.2 Enumeration Type Documentation

2.8.2.1 enum RsaKeyType

Because an asymmetric RSA algorithm is used by Janus to encrypt the AES 256-bit key which symmetrically encrypts a DAVE-ML compliant XML dataset, a calling program must supply file names for the RSA public or private keys. This enum allows a single function to set file names in the Janus instance for both types of keys.

Enumerator:

RSA_PRIVATE_KEY Indicates a RSA private key is being operated on.

RSA_PUBLIC_KEY Indicates a RSA public key is being operated on.

2.8.2.2 enum RsaKeyType `[inherited]`

Because an asymmetric RSA algorithm is used by Janus to encrypt the AES 256-bit key which symmetrically encrypts a DAVE-ML compliant XML dataset, a calling program must supply file names for the RSA public or private keys. This enum allows a single function to set file names in the Janus instance for both types of keys.

Enumerator:

RSA_PRIVATE_KEY Indicates a RSA private key is being operated on.

RSA_PUBLIC_KEY Indicates a RSA public key is being operated on.

2.8.3 Function Documentation

2.8.3.1 int createRsaKeys (void)

This function makes use of the OpenSSL cryptography functions to generate a pair of public and private 2048-bit RSA encryption keys. After creating the keys, it writes them to separate text files, the names of which must previously have been supplied to the Janus instance. Although a Janus instance is required to use this function, it does not require an XML dataset to be supplied to it.

Returns:

Success in generating the keys and writing them to text files results in a 0 return. Various problems can arise at intermediate stages, resulting in JanusErr exceptions being thrown before the function terminates with a return value of 1.

2.8.3.2 int createRsaKeys (void) `[inherited]`

This function makes use of the OpenSSL cryptography functions to generate a pair of public and private 2048-bit RSA encryption keys. After creating the keys, it writes them to separate text files, the names of which must previously have been supplied to the Janus instance. Although a Janus instance is required to use this function, it does not require an XML dataset to be supplied to it.

Returns:

Success in generating the keys and writing them to text files results in a 0 return. Various problems can arise at intermediate stages, resulting in JanusErr exceptions being thrown before the function terminates with a return value of 1.

2.8.3.3 int setRsaKeyFileName (const char ∗ *fileName*, const RsaKeyType *keyType*)

A calling program must supply public or private RSA keys to a Janus instance which is operating on an encrypted dataset. Because the keys are relatively long strings of extremely random characters, they are accessed through text files rather than passed directly to the instance. This function allows the calling program to supply a key file name to the Janus instance, and to distinguish between public and private key file names. Note that a Janus object must be instantiated to use this function, but need not have an XML dataset loaded. NB If an encrypted dataset is to be loaded, an empty Janus instance must be created and the private key file name MUST be loaded BEFORE the XML dataset is loaded with Janus::setXmlFileName.

Parameters:

fileName is the relevant key file name, e.g. "~/pika/Pika+Stores_Public.key"

keyType is an RsaKeyType enum which specifies whether the input file name is relevant to a public or a private key.

Returns:

A value of 0 is returned if the supplied file name is successfully set in the Janus instance, 1 otherwise.

2.8.3.4 int setRsaKeyFileName (const char ∗ *fileName*, const RsaKeyType *keyType*) `[inherited]`

A calling program must supply public or private RSA keys to a Janus instance which is operating on an encrypted dataset. Because the keys are relatively long strings of extremely random characters, they are accessed through text files rather than passed directly to the instance. This function allows the calling program to supply a key file name to the Janus instance, and to distinguish between public and private key file names. Note that a Janus object must be instantiated to use this function, but need not have an XML dataset loaded. NB If an encrypted dataset is to be loaded, an empty Janus instance must be created and the private key file name MUST be loaded BEFORE the XML dataset is loaded with Janus::setXmlFileName.

Parameters:

fileName is the relevant key file name, e.g. "~/pika/Pika+Stores_Public.key"

keyType is an RsaKeyType enum which specifies whether the input file name is relevant to a public or a private key.

Returns:

A value of 0 is returned if the supplied file name is successfully set in the Janus instance, 1 otherwise.

2.8.3.5 int writeEncryptedXmlFile (const char ∗ *fileName*)

This function causes the contents of a DOM, containing an aircraft flight model structure, to be encrypted using an AES 256-bit symmetric key and written to the named file. A typical file may be distinguished by a header in the form:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE DAVEfunc SYSTEM "DAVEfunc.dtd">
<xenc:EncryptedData xmlns:xenc="http://www.w3.org/2001/04/xmlenc#"
                    Type="http://www.w3.org/2001/04/xmlenc#Element">
 <xenc:EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc
                                   #tripledes-cbc"/>
```

Parameters:

fileName is a legal, fully qualified file name to which the current contents of the DOM will be written in encrypted XML format, including an RSA-encrypted symmetric key.

Returns:

A value of 0 is returned if the XML file is written successfully.

2.8.3.6 int writeEncryptedXmlFile (const char ∗ *fileName*) [inherited]

This function causes the contents of a DOM, containing an aircraft flight model structure, to be encrypted using an AES 256-bit symmetric key and written to the named file. A typical file may be distinguished by a header in the form:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE DAVEfunc SYSTEM "DAVEfunc.dtd">
<xenc:EncryptedData xmlns:xenc="http://www.w3.org/2001/04/xmlenc#"
                    Type="http://www.w3.org/2001/04/xmlenc#Element">
 <xenc:EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc
                                   #tripledes-cbc"/>
```

Parameters:

fileName is a legal, fully qualified file name to which the current contents of the DOM will be written in encrypted XML format, including an RSA-encrypted symmetric key.

Returns:

A value of 0 is returned if the XML file is written successfully.

3 Janus Namespace Documentation

3.1 janus Namespace Reference

Classes

- class Janus

3.1.1 Detailed Description

The Janus class and all functions within it are included in the janus namespace. The function names used are unlikely to clash with other libraries, but the namespace provides a more certain way of avoiding such clashes. Note that the error JanusErr, returning information regarding various problems with the class, particularly during instantiation, resides in its own januserr namespace.

3.2 januserr Namespace Reference

Classes

- class JanusErr

Variables

- const int JANUS_ERRMAX = 160

3.2.1 Detailed Description

This namespace contains only the JanusErr class, which has a single data element containing a description of the error which generated the relevant instance of the class. This namespace should not be referenced from outside the Janus class, as it has no external uses and may disappear in future versions of the Janus class library.

3.2.2 Variable Documentation

3.2.2.1 const int JANUS_ERRMAX = 160

Maximum length allocated for error messages (2 lines on terminal)

4 Janus Class Documentation

4.1 Janus Class Reference

```
#include <Janus.h>
```

Public Types

- enum AuthorAttribute {
  NAME, ORG, XNS, EMAIL,
  ADDRESS }
- enum Extrapolation {
  NEITHER, MINEX, MAXEX, BOTH,
  XMIN, XMAX, ERROREX }
- enum Interpolation {
  LINEAR, POLY, CSPLINE, LEGENDRE,
  ERRORIN }
- enum MathOp {
  PLUS, MINUS, TIMES, DIVIDE,
  POWER, SIN, COS, TAN,
  CSC, SEC, COT, ARCSIN,
  ARCCOS, ARCTAN, EXP, LN,
  LOG, ABS, EQUAL, NEQ,
  GT, LT, GEQ, LEQ,
  MIN, MAX, PIECEWISE, AND,
  OR, XOR, NOT, FACTORIAL,
  QUOTIENT, REM, FLOOR, CEILING,
  PI, NOOP }
- enum ModificationAttribute {
  MODID, MOD_REFID, MOD_AUTHORNAME, MOD_AUTHORORG,
  MOD_AUTHORXNS, MOD_AUTHOREMAIL, MOD_AUTHORADDRESS, MOD_-DESCRIPTION }
- enum ReferenceAttribute {
  REFID, AUTHOR, TITLE, ACCESSION,
  DATE, HREF, DESCRIPTION }
- enum RsaKeyType { RSA_PRIVATE_KEY, RSA_PUBLIC_KEY }
- enum VariableType {
  FUNCTION, FUNCTION_INTERNAL, FUNCTION_OUTPUT, MATHML,
  MATHML_INTERNAL, MATHML_OUTPUT, ISOUTPUT, ISINPUT,
  ERRORVT }
- enum VersionType { SHORT, LONG }

Public Member Functions

- int applyOutputScaleFactorByIndex (const int index, const double factor)
- int applyOutputScaleFactorByVarID (const char *varID, const double factor)
- int createRsaKeys (void)
- char * getFunctionDefinitionName (const int index)
- char * getFunctionDescription (const int index)
- char * getFunctionName (const int index)
- double getIndependentVariableByIndex (const int indexf, const int indexv)
- double getIndependentVariableDataMax (const int indexf, const int indexv)
- double getIndependentVariableDataMin (const int indexf, const int indexv)
- double getIndependentVariableDataStep (const int indexf, const int indexv)
- char * getIndependentVariableDescription (const int indexf, const int indexv)
- Extrapolation getIndependentVariableExtrapolation (const int indexf, const int indexv)
- Extrapolation getIndependentVariableExtrapolationFlag (const int indexf, const int indexv)
- char * getIndependentVariableID (const int indexf, const int indexv)
- int getIndependentVariableIndex (const int indexf, const char *varID)
- Interpolation getIndependentVariableInterpolation (const int indexf, const int indexv)
- double getIndependentVariableMax (const int indexf, const int indexv)
- double getIndependentVariableMin (const int indexf, const int indexv)
- char * getIndependentVariableName (const int indexf, const int indexv)
- int getIndependentVariableOrder (const int indexf, const int indexv)
- VariableType getIndependentVariableType (const int indexf, const int indexv)
- char * getIndependentVariableUnits (const int indexf, const int indexv)
- char * getJanusVersion (const VersionType versionType)
- int getNumberOfFunctions ()
- int getNumberOfIndependentVariables (const int index)
- int getNumberOfOutputs ()
- int getNumberOfVariables ()
- double getOutputScaleFactorByIndex (const int index)
- double getOutputScaleFactorByVarID (const char *varID)
- char * getOutputVariable (const int index, int)
- double getOutputVariable (const int index)
- double getOutputVariableByVarID (const char *varID)
- char * getOutputVariableDescription (const int index)
- char * getOutputVariableID (const int index)
- int getOutputVariableIndex (const char *varID)
- char * getOutputVariableName (const int index)
- char * getOutputVariableUnits (const int index)
- double getVariableByIndex (const int index)
- double getVariableByVarID (const char *varID)
- char * getVariableDescription (const int index)
- char * getVariableID (const int index)
- int getVariableIndex (const char *varID)
- char * getVariableName (const int index)
- VariableType getVariableType (const int index)
- char * getVariableUnits (const int index)
- char * getXmlFileAuthor (const AuthorAttribute authorAttribute)
- char * getXmlFileCreationDate ()
- char * getXmlFileDescription ()
- char * getXmlFileModification (const int index, const ModificationAttribute modificationAttribute)
- int getXmlFileModificationCount ()

- int getXmlFileModificationExtraDocCount (const int index)
- char * getXmlFileModificationExtraDocRefID (const int index, const int indxRef)
- char * getXmlFileName ()
- char * getXmlFileReference (const int index, const ReferenceAttribute referenceAttribute)
- int getXmlFileReferenceCount ()
- int getXmlFileReferenceIndex (const char *refID)
- char * getXmlFileVersion ()
- Janus (const char *documentName, const bool validate)
- Janus (const char *documentName)
- Janus ()
- int setDomValidation (const bool validate)
- int setIndependentVariableByIndex (const int indexf, const int indexv, const double x)
- int setRsaKeyFileName (const char *fileName, const RsaKeyType keyType)
- int setVariableByID (const char *varID, const double x)
- int setVariableByIndex (const int index, const double x)
- int setXmlFileName (const char *documentName)
- int writeEncryptedXmlFile (const char *fileName)
- int writeXmlFile (const char *fileName)
- ~Janus ()

4.1.1 Detailed Description

A Janus instance holds in its allocated memory the DOM corresponding to a DAVE-ML compliant XML dataset source file, and data structures derived from that DOM. It also provides the functions which allow a calling program to access the DOM and related data structures, including means of evaluating output variable values which are dependent on supplied input variable values.

The documentation for this class was generated from the following files:

- Janus.h
- BreakpointDef.cpp
- Delaunay.cpp
- FileHeader.cpp
- Function.cpp
- GetDescriptors.cpp
- GetValues.cpp
- GriddedTableDef.cpp
- Janus.cpp
- LinearInterpolation.cpp
- Ludcmp.cpp
- PolyInterpolation.cpp
- Security.cpp
- SetMath.cpp
- SetValues.cpp
- Svd.cpp
- UngriddedInterpolation.cpp
- UngriddedTableDef.cpp
- VariableDef.cpp

4.2 JanusErr Class Reference

```
#include <JanusErr.h>
```

Public Member Functions

- JanusErr (const char *description)

4.2.1 Detailed Description

An instance of the JanusErr class contains a string describing the problem within a Janus class which has caused the JanusErr to be instantiated.

4.2.2 Constructor & Destructor Documentation

4.2.2.1 JanusErr (const char * *description*)

This function is used within Janus to throw a potentially informative error message, particularly during initialisation of a Janus instance. The error thrown must be dealt with by the code which instantiates Janus at this point, or the program will abort. This class should not be used for errors outside Janus, as it may disappear from public view in future versions.

Parameters:

description is intended to detail the error encountered.

The documentation for this class was generated from the following files:

- JanusErr.h
- JanusErr.cpp

5 Janus File Documentation

5.1 BreakpointDef.cpp File Reference

```
#include "Janus.h"
```

5.1.1 Detailed Description

This code is used during initialisation of the Janus class, and provides access to the breakpoint definitions contained in a DOM which complies with the DAVE-ML DTD.

A breakpointDef is where gridded table breakpoints are given. Since these are separate from function data, they may be reused.

bpVals is a set of breakpoints; that is, a set of independent variable values associated with one dimension of a gridded table of data. An example would be the Mach or angle-of-attack values that define the coordinates of each data point in a two-dimensional coefficient value table.

Author: D. M. Newman (dmnewman@pobox.com)

Date	By	Effect
18jul03	dmn	initial release
28jun04	dmn	test release of inital capability
19jan05	dmn	added documentation
19feb05	dmn	fixed transcode memory leaks
22feb05	dmn	speed up initialisation string handling
24feb05	dmn	modified for Carna compatibility
10mar05	dmn	added namespace
05jul05	dmn	loop variable external allocation

5.2 Delaunay.cpp File Reference

```
#include <cstdio>
#include "Janus.h"
#include <qhull/qhull.h>
#include <qhull/mem.h>
#include <qhull/qset.h>
#include <qhull/geom.h>
#include <qhull/merge.h>
#include <qhull/poly.h>
#include <qhull/io.h>
#include <qhull/stat.h>
```

5.2.1 Detailed Description

This private function generates multi-dimensional Delaunay triangulations based on ungridded datasets, using the Qhull library. It is called by `setUngriddedTableDefsFromDom`. To avoid problems with distribution of the Janus code, it is normally statically linked with the Qhull library.

Author: D. M. Newman (dmnewman@pobox.com)

Date	By	Effect
27jan05	dmn	initial release
25feb05	dmn	fixed memory leak
10mar05	dmn	added namespace
05jul05	dmn	loop variable external allocation

5.3 FileHeader.cpp File Reference

```
#include <cstdio>
```

```
#include "Janus.h"
```

5.3.1 Detailed Description

The 'fileHeader' element requires an author, a creation date and a version indicator; optional content are description, references and mod records.

This function sets up pointers to transcoded versions of the strings contained in the header element.

Author: D. M. Newman (dmnewman@pobox.com)

Date	By	Effect
25mar05	dmn	initial release
05jul05	dmn	loop variable external allocation

5.4 Function.cpp File Reference

```
#include <cstdio>
```

```
#include <cmath>
```

```
#include "Janus.h"
```

5.4.1 Detailed Description

This code is used during initialisation of the Janus class, and provides access to the function definitions contained in a DOM which complies with the DAVE-ML DTD.

Each function has optional description, optional provenance, and either a simple input/output values or references to more complete (possible multiple) input, output, and function data elements.

Author: D. M. Newman (dmnewman@pobox.com)

Date	By	Effect
18jul03	dmn	initial release
28jun04	dmn	test release of inital capability
08dec04	dmn	added interpolation type data access
20dec04	dmn	separate outputs from functions
18jan05	dmn	remove acos NaN generation & add variable
20jan05	dmn	initial ungridded function components
17feb05	dmn	changed test for unlabelled internal tables
19feb05	dmn	fixed transcode memory leaks
22feb05	dmn	speed up initialisation string handling
23feb05	dmn	changed variable type flags for compatibility with Carna implicit functions
24feb05	dmn	further modified for Carna compatibility
10mar05	dmn	added namespace
10mar05	dmn	for min/max, return either end breakpoints or ungridded extrema if not defined explicitly
13mar05	dmn	data min and max function values added
05jul05	dmn	loop variable external allocation

5.5 GetDescriptors.cpp File Reference

```
#include <cmath>
#include "Janus.h"
```

5.5.1 Detailed Description

This code is used during interrogation of an instance the Janus class, and provides the calling program access to the descriptive elements contained in a DOM which complies with the DAVE-ML DTD.

In keeping with the data's descriptive nature, most returns from these functions are strings, although there are a few numerical values and an enum.

Author: D. M. Newman (dmnewman@pobox.com)

Date	By	Effect
18jul03	dmn	initial release
28jun04	dmn	test release of inital capability
08dec04	dmn	added interpolation type access functions
20dec04	dmn	separate outputs from functions
14jan05	dmn	add mathml related and type functions
18jan05	dmn	remove acos NaN generation
10mar05	dmn	returns version description
10mar05	dmn	added namespace
13mar05	dmn	data min and max function values added
28mar05	dmn	header details functions added
05jul05	dmn	loop variable external allocation

5.6 GetValues.cpp File Reference

```
#include <cmath>
#include "Janus.h"
```

5.6.1 Detailed Description

This code is used at run-time to determine output variable values, based on independent variable values supplied to the instanced Janus class.

It applies to linear or polynomial interpolation of gridded data, constant outputs, or evaluation of MathML expressions. Through this code the source and type of data is made irrelevant to the calling program.

Other function models and evaluation techniques will be included in future releases.

Author: D. M. Newman (dmnewman@pobox.com)

Date	By	Effect
18jul03	dmn	initial release
28jun04	dmn	test release of inital capability
10oct04	dmn	limit of 32 dof after sdh's tests
20dec04	dmn	separate outputs from functions
17jan05	dmn	initial mathml functionality
18jan05	dmn	remove acos NaN generation
20jan05	dmn	initial ungridded table structures
10feb05	dmn	changed error handling
22feb05	dmn	fixed transcode memory leaks
10mar05	dmn	added namespace
22Jun05	dmn	string table handling
05jul05	dmn	loop variable external allocation

5.7 GriddedTableDef.cpp File Reference

```
#include <cstdio>
#include <cctype>
#include "Janus.h"
```

5.7.1 Detailed Description

This code is used during initialisation of the Janus class, and provides access to gridded table definitions contained in a DOM which complies with the DAVE-ML DTD.

A griddedTableDef contains points arranged in an orthogonal (but multi-dimensional) array, where the independent variables are defined by separate breakpoint vectors. This table definition is specified separately from the actual function declaration and requires an XML identifier attribute so that it may be used by multiple functions. The table data point values are specified as comma-separated values in floating-point notation (0.93638E-06) in a single long sequence as if the table had been unraveled with the last-specified dimension changing most rapidly. Line breaks are to be ignored. Comments may be embedded in the table to promote [human] readability.

Author: D. M. Newman (dmnewman@pobox.com)

Date	By	Effect
18jul03	dmn	initial release
28jun04	dmn	test release of inital capability
07dec04	dmn	transfer data separately to each referring function
19jan05	dmn	added documentation
24jan05	dmn	changed griddedTable handing to match new ungridded form
19feb05	dmn	fixed transcode memory leaks
22feb05	dmn	speed up initialisation string handling
10mar05	dmn	added namespace
23May05	dmn	sdh's changes for move gridded data to table
22Jun05	dmn	string table handling
05jul05	dmn	loop variable external allocation

5.8 Janus.cpp File Reference

```
#include <cstdio>
#include <iostream>
#include "Janus.h"
```

5.8.1 Detailed Description

Includes constructors, destructor, and related private functions. Can perform XML initialisation and instance an Xerces parser, then load from the supplied XML file to a DOM structure. Holds the structure and accesses it on request, doing interpolation and function evaluation as required for output. Cleans up on termination.

Author: D. M. Newman (dmnewman@pobox.com)

Date	By	Effect
18jul03	dmn	initial release
28jun04	dmn	test release of inital capability
20dec04	dmn	separate outputs from functions
12jan04	dmn	initial mathml references, change DOMBuilder setFeature entries
14jan04	dmn	set up any computed constants
20jan05	dmn	initial ungridded table components
11feb05	dmn	inital Store-ML components
15feb05	dmn	Store-ML components separated
16feb05	dmn	set default validate off, improved error msg
19feb05	dmn	fixed transcode memory leaks
24feb05	dmn	modified for Carna compatibility
12mar05	dmn	because of DAVE-ML / MathML uncertainty, always load external DTD if available. This will need to be changed later
13mar05	dmn	data min and max function values added
25mar05	dmn	fileHeader initial components added
22Jun05	dmn	string table handling, uninitialised workspace flag
05jul05	dmn	loop variable external allocation
13jul05	dmn	XML security added

5.9 Janus.h File Reference

```
#include <cstdlib>
#include <cstring>
#include <xercesc/parsers/XercesDOMParser.hpp>
#include <xercesc/dom/DOM.hpp>
#include <xercesc/sax/HandlerBase.hpp>
#include <xercesc/util/XMLString.hpp>
#include <xercesc/util/PlatformUtils.hpp>
#include <xercesc/framework/LocalFileFormatTarget.hpp>
#include "JanusConfig.h"
#include "JanusErr.h"
#include "JanusSecurity.h"
```

Namespaces

- namespace janus

Classes

- class Janus

5.9.1 Detailed Description

Janus performs XML initialisation and instances an Xerces parser, then loads from the supplied XML file to a DOM structure. It holds the structure and accesses it on request, doing interpolation or other computation as required for output. It cleans up on termination.

This header defines all the elements required to use the XML dataset for flight modelling, and should be included in any source code intended to activate an instance of the Janus class.

This header also contains documentation which forms the basis of a doxygen-generated manual.

Author: D. M. Newman (`dmnewman@pobox.com`)

Date	By	Effect
18jul03	dmn	initial release
28jun04	dmn	test release of inital capability
10oct04	dmn	limit of 32 dof after sdh's tests
08dec04	dmn	polynomial interpolation
20dec04	dmn	separate outputs from functions
11jan05	dmn	static elements for polynomials, update documenatation
12jan05	dmn	initial mathml structures, changed outputType to variableDefType
18jan05	dmn	remove acos NaN generation
18jan05	dmn	pre-parse and store MathML functions
20jan05	dmn	initial ungridded table structures
10feb05	dmn	MathML logical elements
11feb05	dmn	extended MathML logical elements
11feb05	dmn	initial Store-ML elements
15feb05	dmn	Store-ML components separated
20feb05	dmn	fixed transcode memory leaks
23feb05	dmn	changed variable type flags for compatibility with Carna implicit functions
10mar05	dmn	added global versions, namespace
13mar05	dmn	data min and max function values added
25mar05	dmn	fileHeader initial components added
21apr05	dmn	documentation additions and corrections
20jun05	dmn	string table handling
29jun05	dmn	sdh's NAN macros
05jul05	dmn	minor cleanups
13jul05	dmn	XML security added

5.10 JanusErr.cpp File Reference

```
#include <cstring>
#include "JanusErr.h"
```

5.10.1 Detailed Description

This is a class for Janus exceptions of all types. It displays a message (possibly useful) if something happens which would cause problems in any executable element of the class. These errors are generally thrown in the initialisation stages of a Janus instance, where most checking occurs.

Author: D. M. Newman (dmnewman@pobox.com)

Date	By	Effect
10jan04	dmn	initial release
28jun04	dmn	test release of inital capability
10oct04	dmn	changed for uniformity with main program
19jan05	dmn	added documentation
10mar05	dmn	added namespace
05jul05	dmn	minor cleanups

5.11 JanusErr.h File Reference

Namespaces

- namespace januserr

Classes

- class JanusErr

Variables

- const int januserr::JANUS_ERRMAX = 160

5.11.1 Detailed Description

Handles errors occurring during the Janus instantiation process. These mostly relate to the ability (or otherwise) to find XML files, DTD files, and perform loading and validation of the DOM. A failure in this area will cause a JanusErr to be thrown, which must be handled by the calling function to avoid a program abort.

Author: D. M. Newman (dmnewman@pobox.com)

Date	By	Effect
10jan04	dmn	initial release
28jun04	dmn	test release of inital capability
10oct04	dmn	changed for uniformity with main program
14dec04	dmn	warn of possible future removal
19jan05	dmn	added documentation
10mar05	dmn	added namespace
05jul05	dmn	minor cleanups

5.12 JanusSecurity.h File Reference

Enumerations

- enum RsaKeyType { RSA_PRIVATE_KEY, RSA_PUBLIC_KEY }

Functions

- int createRsaKeys (void)
- int setRsaKeyFileName (const char *fileName, const RsaKeyType keyType)
- int writeEncryptedXmlFile (const char *fileName)

5.12.1 Detailed Description

This header defines the extra private elements required to use encryption and decryption of XML datasets for flight modelling, and is included at the end of the main 'Janus.h' header. It therefore does not need to be explicitly included by any other file.

This header also contains documentation which forms the basis of a doxygen-generated manual.

Author: D. M. Newman (dmnewman@pobox.com)

Date	By	Effect
13jul05	dmn	initial release

5.13 LinearInterpolation.cpp File Reference

```
#include "Janus.h"
```

5.13.1 Detailed Description

This private function performs interpolations when all the degrees of freedom for a function are specified as linear or first order polynomial, or for the default condition when interpolationType is not specified.

Given $2^\wedge n$ uniformly gridded values of a function of n variables, provided to the instance of the class by either `setVariableByIndex` or `setVariableByID`, this private function is called by `getOutputVariable` to perform a multi-linear interpolation between the values and returns the result. It maintains continuity of function across the grid, but not of derivatives of the function. NB if the fractions based on the grid direction variables are outside the range 0.0 -> 1.0 this function can perform an extrapolation, controlled by the 'extrapolate' attribute, with possibly dubious results depending on the shape of the represented function.

Author: D. M. Newman (dmnewman@pobox.com)

Date	By	Effect
18jul03	dmn	initial release
28jun04	dmn	test release of inital capability
10oct04	dmn	limit of 32 dof after sdh's tests
08dec04	dmn	separated from getOutputVariable to allow polynomial interpolation as an option
14dec04	dmn	adjusted limit extrapolation behaviour
20dec04	dmn	separate outputs from functions
18jan05	dmn	removed NaN test for limits
10mar05	dmn	added namespace

5.14 Ludcmp.cpp File Reference

```
#include <cmath>
#include "Janus.h"
```

5.14.1 Detailed Description

These procedures perform a double precision L-U decomposition and back-substitution, adapted from "Numerical Recipes - The Art of Scientific Computing in Fortran" by Press et al, converted to C code and to work in double precision.

Author: D. M. Newman (dmnewman@pobox.com)

Date	By	Effect
14may94	dmn	initial release
03feb05	dmn	adjusted for use with Janus
10mar05	dmn	added namespace
05jul05	dmn	minor cleanups

5.15 PolyInterpolation.cpp File Reference

```
#include <cstdio>
#include <cmath>
#include "Janus.h"
```

5.15.1 Detailed Description

This private function performs interpolations when *not* all the degrees of freedom for a function are specified as linear or first order polynomial.

If the interpolation order in the i th degree of freedom is k_i , then given $\Pi_1^n(k_i + 1)$ uniformly gridded values of a function of n variables, provided to the instance of the class by either `setVariableByIndex` or `setVariableByID`, this private function is called by `getOutputVariable` to perform a multi-dimensional polynomial interpolation between the values and returns the result. At present the maximum polynomial order is limited to 3. The interpolation maintains continuity of function across the grid, but not of derivatives of the function. NB this function can perform an extrapolation, controlled by the 'extrapolate' attribute, but polynomial extrapolation is notoriously inaccurate and unstable and should not be relied on by any user interested in maintaining modelling fidelity. You have been warned ...

Author: D. M. Newman (dmnewman@pobox.com)

Date	By	Effect
08dec04	dmn	initial release, not functional
14dec04	dmn	adjusted limit extrapolation behaviour
20dec04	dmn	separate outputs from functions
11jan05	dmn	revised max order to 3, reduced higher order computation requirement slightly, changed higher order breakpoint offset
10jan05	dmn	revised offset computation
18jan05	dmn	removed NaN test for limits
10feb05	dmn	adjusted limit test
10mar05	dmn	added namespace
05jul05	dmn	loop variable external allocation

5.16 Security.cpp File Reference

```
#include <cstdio>
#include <iostream>
#include "Janus.h"
```

5.16.1 Detailed Description

Aditional components required to handle XML dataset encryption and decryption. These make use of the XML-Security-C library from Apache to perform the DOM handling aspects, and the actual encryption and decryption is handled by the OpenSSL library.

The OpenSSL library includes cryptographic software written by Eric Young (eay@cryptsoft.com), and the copyright of that library remains Eric Young's.

Author: D. M. Newman (dmnewman@pobox.com)

Date	By	Effect
14jul05	dmn	initial release
19jul05	dmn	explicitly seed RNG for RSA key generation

5.17 SetMath.cpp File Reference

```
#include <cstdio>
#include <cmath>
#include "Janus.h"
```

5.17.1 Detailed Description

'variableDef' elements can include MathML content markup to indicate any calculation required to arrive at the value of the variable, using other variables as inputs. Such content is enclosed in 'calculation' tags.

These functions parse the contents of 'calculation' elements within the DOM and generate function trees based on the contents, which are then accessed whenever an output value for a 'calculation'-based variable is requested. Each 'math' element has an 'apply' child which itself has as children an operator (eg 'plus', 'times') followed by the input(s) to the operator as either a number (<cn>), a variable (<ci>), or the result of another computation (<apply>).

At present it is assumed that all DOM MathML data consists of real numbers, which are loaded and processed in double precision.

Author: D. M. Newman (dmnewman@pobox.com)

Date	By	Effect
18jan05	dmn	initial release
08feb05	dmn	added new functions
10feb05	dmn	MathML logical elements
11feb05	dmn	extended MathML logical elements
17feb05	dmn	prevent reference to uninitialised variableDefs
20feb05	dmn	fixed transcode memory leaks
22feb05	dmn	speed up initialisation string handling
10mar05	dmn	added namespace
20jun05	dmn	moved piece tags out of loop
05jul05	dmn	minor cleanups

5.18 SetValues.cpp File Reference

```
#include "Janus.h"
```

5.18.1 Detailed Description

Trivial, but very necessary. These functions provide various ways to set the values of the independent variables which are used in the function evaluations, and to apply scale factors to tabulated function data.

Author: D. M. Newman (dmnewman@pobox.com)

Date	By	Effect
18jul03	dmn	initial release
28jun04	dmn	test release of inital capability
20dec04	dmn	separate outputs from functions
20dec04	dmn	added documentation
20jan05	dmn	initial ungridded table structures
22feb05	dmn	fixed transcode memory leaks
10mar05	dmn	added namespace
05jul05	dmn	loop variable external allocation

5.19 Svd.cpp File Reference

```
#include <cmath>
#include "Janus.h"
```

5.19.1 Detailed Description

These procedures are adapted from "Numerical Recipes - The Art of Scientific Computing in Fortran" by Press et al, converted to C code and to work in double precision.

Author: D. M. Newman (dmnewman@pobox.com)

Date	By	Effect
15apr99	dmn	initial release
29apr99	dmn	added dummy statement after L3 for compiler requiring execution after label
03feb05	dmn	adjusted for use with Janus
10mar05	dmn	added namespace
05jul05	dmn	minor cleanups

5.20 UngriddedInterpolation.cpp File Reference

```
#include <cstdio>
#include <cmath>
#include "Janus.h"
```

5.20.1 Detailed Description

This private function performs linear interpolations on ungridded datasets. It is called by `getOutputVariable` to perform a multi-linear interpolation between the values and returns the result. It maintains continuity of function across the dataset, but not of derivatives of the function.

Author: D. M. Newman (dmnewman@pobox.com)

Date	By	Effect
21jan05	dmn	initial release
10mar05	dmn	added namespace
05jul05	dmn	loop variable external allocation

5.21 UngriddedTableDef.cpp File Reference

`#include "Janus.h"`

5.21.1 Detailed Description

This code is used during initialisation of the Janus class, and provides access to ungridded table definitions contained in a DOM which complies with the DAVE-ML DTD.

An ungriddedTableDef contains points that are not in an orthogonal grid pattern; thus, the independent variable coordinates are specified for each dependent variable value. This table definition is specified separately from the actual function declaration and requires an XML identifier attribute so that it may be used by multiple functions.

Author: D. M. Newman (dmnewman@pobox.com)

Date	By	Effect
20jan05	dmn	initial release
19feb05	dmn	fixed transcode memory leaks
22feb05	dmn	speed up initialisation string handling
10mar05	dmn	added namespace
05jul05	dmn	loop variable external allocation

5.22 VariableDef.cpp File Reference

`#include <cstdio>`

`#include <cmath>`

`#include "Janus.h"`

5.22.1 Detailed Description

'variableDef' elements provide wiring information - that is, they identify the input and output signals used by these function blocks. They also provide MathML content markup to indicate any calculation required to arrive at the value of the variable, using other variables as inputs. The variable definition can include statistical information regarding the uncertainty of the values which it might take on, when measured after any calculation is performed.

This function also allocates memory for an index of output variables and collates those with either 'calculation' or 'isOutput' nodes. setFunctionsFromDom can later add to this list any dependent variables which are not explicitly flagged as outputs.

Author: D. M. Newman (dmnewman@pobox.com)

Date	By	Effect
18jul03	dmn	initial release
28jun04	dmn	test release of inital capability
20dec04	dmn	separate outputs from functions
12jan05	dmn	initial mathml output indices
19jan05	dmn	function cross-reference indices
17feb05	dmn	prevent reference to uninitialised variableDefs
19feb05	dmn	fixed transcode memory leaks
22feb05	dmn	speed up initialisation string handling
10mar05	dmn	added namespace
05jul05	dmn	loop variable external allocation

References

[Newman, 2004] D. M. Newman, *Efficient Development and Use of Aircraft Flight Models – Survey of DSTO Usage*, Ball Solutions Group Report No. 1234.002, Melbourne, 2004.

[AIAA, 2003] Anon., *Standards for the Exchange of Simulation Modeling Data*, AIAA Modeling and Simulation Technical Committee, Preliminary Draft, January 2003.

[Jackson & Hildreth, 2002] E. B. Jackson & B. L. Hildreth, *Flight Dynamic Model Exchange using XML* AIAA 2002-4482, AIAA Modeling and Simulation Technologies Conference, Monterey, CA, August 2002.

[AIAA, 2004] Anon., *Dynamic Aerospace Vehicle Exchange Markup Language (DAVE-ML) Reference*, AIAA Simulation Standards Working Group, Version 1.7b1, viewed 20 December 2004, `http://daveml.nasa.gov/DTDs/dev/DAVE-ML_ref.pdf`.

[AIAA, 2001] Anon., *Standard Simulation Variable Names*, AIAA Flight Simulation Technical Committee, December 2001.

[Brian, 2004] G. Brian, *Flight Systems Units of Measure Guidelines*, viewed 12 December 2004, `http://larda/Reference/UnitStandard/FltSysUnitStandard.html`.

1 1, 13, 14 1 1 20, 25, 38