# SPARK Atomic Class API Reference Manual

VisualSPARK 2.01

by Dimitri Curtil

Wed Nov 5 14:54:57 2003

# Contents

# Chapter 1

# The SPARK Atomic Class API

This document describes the preprocessor macro definitions, classes and functions that are used to implement the inverses in the user-defined atomic classes. The macros and functions are declared in the header file spark.h. After installation of the VisualSPARK release package, the header file spark.h can be found in the `inc/` subdirectory in the main SPARK directory.

The header file spark.h should be included in every file where user-defined atomic classes are implemented to ensure that the required declarations are provided correctly at compile-time.

More detailed information about how to write an atomic class can be found in:

- the SPARK Reference Manual
- the VisualSPARK Users Guide
- the VisualSPARK tutorial

Visit `http://simulationresearch.lbl.gov/` to obtain the latest versions of these documents.

## 1.1 List of preprocessor macros

Macros to declare the callback prototypes:

- NON_MODIFIER_CALLBACK( func )
- MODIFIER_CALLBACK( func )
- PREDICATE_CALLBACK( func )
- STATIC_NON_MODIFIER_CALLBACK( func )
- STATIC_PREDICATE_CALLBACK( func )

Macros to declare each callback type:

- EVALUATE( func )
- PREDICT( func )
- CONSTRUCT( func )
- PREPARE_STEP( func )
- CHECK_INTEGRATION_STEP( func )

- COMMIT( func )
- ROLLBACK( func )
- DESTRUCT( func )
- STATIC_CONSTRUCT( func )
- STATIC_PREPARE_STEP( func )
- STATIC_CHECK_INTEGRATION_STEP( func )
- STATIC_COMMIT( func )
- STATIC_ROLLBACK( func )
- STATIC_DESTRUCT( func )

Macros to manage private data associated with each object and/or inverse:

- THIS
- GET_DATA
- SET_DATA
- DELETE_DATA
- ACTIVE_PROBLEM
- ACTIVE_INVERSE
- ACTIVE_COMPONENT

Macros to manage the argument variables passed to a callback:

- ARGUMENT( n, Name )
- ARGDEF( n , Name )

Macros to manage the target variables returned from a callback:

- TARGET( n, Name )
- RETURN( val )

Macros to send a problem request from an atomic class:

- REQUEST__ABORT( context )
- REQUEST__STOP( context )
- REQUEST__SET_STOP_TIME( context, time )
- REQUEST__REPORT( context )
- REQUEST__SNAPSHOT( context, filename )
- REQUEST__SET_MEETING_POINT( context, time )
- REQUEST__CLEAR_MEETING_POINTS( context )
- REQUEST__RESTART( context )
- REQUEST__SET_TIME_STEP( context, h )

## 1.2 List of classes and typedefs

The following classes:

- SPARK::TVariable
- SPARK::TUnknown
- SPARK::TArgument
- SPARK::TTarget

used to represent the problem variables in the SPARK solver have also been documented. They are defined in the header files variable.h and callback.h which can be found in the `inc/` subdirectory.

Finally, the classes that implement the callbacks, inverses and objects are described:

- SPARK::TInverse
- SPARK::TObject
- SPARK::TModifierCallback
- SPARK::TNonModifierCallback
- SPARK::TPredicateCallback
- SPARK::TStaticNonModifierCallback
- SPARK::TStaticPredicateCallback

# Chapter 2

# SPARK Atomic Class API Namespace Documentation

## 2.1 SPARK Namespace Reference

Definitions of numerical constants, math functions, and the various types used to describe a SPARK problem.

### Classes

- class TVariable

  *Class used to represent the properties and the numerical values of a problem variable.*

- class TUnknown

  *Class used to represent the properties and the numerical values of an unknown problem variable.*

- class TInverse

  *Class that defines the callbacks for an inverse.*

- class TObject

  *Class used to represent an instance of an inverse.*

- class TArgument

  *This class acts as a read-only interface to a TVariable object. It is used only in the callbacks to describe the argument variables. This class is not used internally in the solver. It provides proper value access behavior depending on the type of the variable and the calling context. Also, it implements some of the attribute access methods from TVariable.*

- class TTarget

  *This class acts as a write-only interface to a TVariable object. It is used only in the callbacks to describe the target variables. This class is not used internally in the solver. It provides proper value access behavior depending on the type of the variable and the calling context. Also, it implements some of the attribute access methods from TVariable.*

- class TFreeFunctionWrapper

  *Wrapper class for any free function that is invoked as an atomic class callback.*

- struct TEnumPolicy

  *Policy class that specifies a enum value of type EnumType.*

- class TModifierCallback

  *Function wrapper class for modifier callbacks.*

- class TNonModifierCallback

  *Function wrapper class for non-modifier callbacks.*

- class TPredicateCallback

  *Function wrapper class for predicate callbacks.*

- class TStaticNonModifierCallback

  *Function wrapper class for static non-modifier callbacks.*

- class TStaticPredicateCallback

  *Function wrapper class for static predicate callbacks.*

- class TProblem

  *Representation of a problem object in the SPARK solver.*

- class TProblem::TState

  *Interface class defining the methods used to save and restore the state of the problem using the TProblem::Save() and TProblem::Restore() methods.*

- class TComponent

  *Class that solves the set of DAE equations generated by setupcpp.*

- struct delete_policy

  *Policy class used to delete non-array data types using delete operator.*

- struct delete_array_policy

  *Policy class used to delete array data types using delete [] operator.*

- struct deleter

  *Helper class that implements either delete_policy or delete_array_policy depending on the compile-time value of the non-type parameter IsArray.*

- class XAssertion

  *Base class for all SPARK exceptions.*

- class XDimension

  *Indicates that a runtime error occured due to mismatched dimension.*

- class XOutOfRange

  *Indicates that a runtime error occured due to an out of range access operation on a container.*

- class XMemory

  *Indicates that a runtime error occured because memory could not be allocated.*

- class XInitialization

  *Indicates that a runtime error occured while initializing an object.*

- class XIO

  *Indicates that a runtime error occured while performing an IO operation.*

- class XTimeStep

  *Indicates that a runtime error occured while adapting the time step.*

- class XStepper

  *Inidicates that stepping to the next step failed.*

## Functions used inside the solver code

- enum NumericalValueTypes {
  NumericalValueType_VALID = 0,
  NumericalValueType_INF = 1,
  NumericalValueType_NAN = 2 }

  *Codes returned by SPARK::InfiniteOrNaN() function.*

- template<typename T> NumericalValueTypes InfiniteOrNaN (T scalar)

  *Checks whether the floating-point number* `scalar` *is infinite or NaN or a valid numerical scalar.*

## Functions used in HVAC toolkit

- double abs (double x)
- double min (double x, double y)
- double max (double x, double y)
- double sign (double x)

  *Returns the +1.0 if* `x` *is positive, -1.0 otherwise.*

- double sign (double retval, double test)

  *Returns the* `a` *if* `b` *is strictly positive,* `-a` *otherwise.*

- double log2 (double x)

## Constant Declaration

- const double TINY = 1.0E-30

  *Considered essentially as zero in solver (e.g., used to detect singularity).*

- const double SQRT_UROUND = sqrt(UROUND)

  *Square root of the unit round-off error.*

## Typedefs

- typedef SPARK::container< SPARK::TArgument > TArguments

  *Type for collection of TArgument objects.*

- typedef const TArguments & ArgList

  *Type of argument list passed to the non-static callback functions.*

- typedef SPARK::container< SPARK::TTarget > TTargets

  *Type for collection of TTarget objects.*

- typedef TTargets & TargetList

  *Type of target list passed to the the modifier callback functions.*

- typedef void(* TModifierFunction )(TObject *, ArgList, TargetList)

  *Function prototype for modifier callbacks.*

- typedef void(* TNonModifierFunction )(TObject *, ArgList)

  *Function prototype for non-modifier callbacks.*

- typedef bool(* TPredicateFunction )(TObject *, ArgList)

  *Function prototype for predicate callbacks.*

- typedef void(* TStaticNonModifierFunction )(TInverse *)

  *Function prototype for static callbacks.*

- typedef bool(* TStaticPredicateFunction )(TInverse *)

  *Function prototype for predicate callbacks.*

## Enumerations

- enum CallbackTypes {
  CallbackType_EVALUATE = 0,
  CallbackType_PREDICT,
  CallbackType_CONSTRUCT,
  CallbackType_DESTRUCT,
  CallbackType_PREPARE_STEP,
  CallbackType_ROLLBACK,
  CallbackType_COMMIT,
  CallbackType_CHECK_INTEGRATION_STEP,
  CallbackType_STATIC_CONSTRUCT,
  CallbackType_STATIC_DESTRUCT,
  CallbackType_STATIC_PREPARE_STEP,
  CallbackType_STATIC_ROLLBACK,
  CallbackType_STATIC_COMMIT,
  CallbackType_STATIC_CHECK_INTEGRATION_STEP,
  CALLBACKTYPES_L,
  CallbackType_NONE = CALLBACKTYPES_L }

  *Enum for callback types.*

- enum ProtoTypes {
  ProtoType_MODIFIER = 0,
  ProtoType_NON_MODIFIER,
  ProtoType_PREDICATE,
  ProtoType_STATIC_NON_MODIFIER,
  ProtoType_STATIC_PREDICATE,
  PROTOTYPES_L,
  ProtoType_NONE = PROTOTYPES_L }

  *Enum for callback prototypes.*

- enum ReturnTypes {
  ReturnType_VALUE = 0,
  ReturnType_RESIDUAL,
  RETURNTYPES_L,
  ReturnType_NONE = RETURNTYPES_L }

  *Enum for return types from modifier callbacks.*

- enum VariableTypes {
  VariableType_GLOBAL_TIME = 0,
  VariableType_GLOBAL_TIME_STEP,
  VariableType_PARAMETER,
  VariableType_INPUT,
  VariableType_UNKNOWN,
  VARIABLETYPES_L,
  VariableType_NONE = VARIABLETYPES_L }

  *Enum for the various variable types in the problem.*

- enum RequestTypes {
  RequestType_ABORT = 0,
  RequestType_STOP,
  RequestType_SET_STOP_TIME,
  RequestType_REPORT,
  RequestType_SNAPSHOT,
  RequestType_SET_MEETING_POINT,
  RequestType_CLEAR_MEETING_POINTS,
  RequestType_RESTART,
  RequestType_SET_TIME_STEP,
  RequestType_SET_DYNAMIC_STEPPER,
  REQUESTTYPES_L,
  RequestType_NONE = REQUESTTYPES_L }

## Functions

- std::ostream & operator<< (std::ostream &os, const SPARK::TUnknown &unknown)
  *Output operator << overload for the SPARK::TUnknown class.*
- std::ostream & operator<< (std::ostream &os, const SPARK::TVariable &V)
  *Output operator << overload for the SPARK::TVariable class.*
- std::ostream & operator<< (std::ostream &os, const SPARK::TArgument &argument)
  *Output operator << overload for the TArgument class.*
- std::ostream & operator<< (std::ostream &os, const SPARK::TTarget &target)
  *Output operator << overload for the TTarget class.*

### 2.1.1 Detailed Description

Definitions of numerical constants, math functions, and the various types used to describe a SPARK problem.

### 2.1.2 Typedef Documentation

#### 2.1.2.1 typedef SPARK::container< SPARK::TArgument > SPARK::TArguments

Type for collection of TArgument objects.

#### 2.1.2.2 typedef const TArguments& SPARK::ArgList

Type of argument list passed to the non-static callback functions.

#### 2.1.2.3 typedef SPARK::container< SPARK::TTarget > SPARK::TTargets

Type for collection of TTarget objects.

#### 2.1.2.4 typedef TTargets& SPARK::TargetList

Type of target list passed to the the modifier callback functions.

#### 2.1.2.5 typedef void(* SPARK::TModifierFunction)(TObject* , ArgList , TargetList )

Function prototype for modifier callbacks.

#### 2.1.2.6 typedef void(* SPARK::TNonModifierFunction)(TObject* , ArgList )

Function prototype for non-modifier callbacks.

#### 2.1.2.7 typedef bool(* SPARK::TPredicateFunction)(TObject* , ArgList )

Function prototype for predicate callbacks.

#### 2.1.2.8 typedef void(* SPARK::TStaticNonModifierFunction)(TInverse* )

Function prototype for static callbacks.

#### 2.1.2.9 typedef bool(* SPARK::TStaticPredicateFunction)(TInverse* )

Function prototype for predicate callbacks.

### 2.1.3 Enumeration Type Documentation

#### 2.1.3.1 enum SPARK::NumericalValueTypes

Codes returned by SPARK::InfiniteOrNaN() function.

**Enumeration values:**

***NumericalValueType_VALID*** numerical value is valid

***NumericalValueType_INF*** numerical value is positive infinite

***NumericalValueType_NAN*** numerical value is not a number

#### 2.1.3.2 enum SPARK::CallbackTypes

Enum for callback types.

**Enumeration values:**

***CallbackType_EVALUATE*** Specifies an EVALUATE callback.

***CallbackType_PREDICT*** Specifies a PREDICT callback.

***CallbackType_CONSTRUCT*** Specifies a CONSTRUCT callback.

***CallbackType_DESTRUCT*** Specifies a DESTRUCT callback.

***CallbackType_PREPARE_STEP*** Specifies a PREPARE_STEP callback.

***CallbackType_ROLLBACK*** Specifies a ROLLBACK callback.

***CallbackType_COMMIT*** Specifies a COMMIT callback.

***CallbackType_CHECK_INTEGRATION_STEP*** Specifies a CHECK_INTEGRATION_STEP callback for CLASSTYPE=INTEGRATOR only.

***CallbackType_STATIC_CONSTRUCT*** Specifies a STATIC_CONSTRUCT callback.

***CallbackType_STATIC_DESTRUCT*** Specifies a STATIC_DESTRUCT callback.

***CallbackType_STATIC_PREPARE_STEP*** Specifies a STATIC_PREPARE_STEP callback.

***CallbackType_STATIC_ROLLBACK*** Specifies a STATIC_ROLLBACk callback.

***CallbackType_STATIC_COMMIT*** Specifies a STATIC_COMMIT callback.

***CallbackType_STATIC_CHECK_INTEGRATION_STEP*** Specifies a STATIC_CHECK_INTEGRATION_STEP callback for CLASSTYPE=INTEGRATOR only.

***CALLBACKTYPES_L*** Number of different callback types.

***CallbackType_NONE*** Default value for non-initialized CallbackTypes variables.

#### 2.1.3.3 enum SPARK::ProtoTypes

Enum for callback prototypes.

**Enumeration values:**

***ProtoType_MODIFIER*** Prototype used by modifier callbacks (e.g., evaluate and predict).

***ProtoType_NON_MODIFIER*** Prototype used by non-modifier callbacks (e.g., construct, destruct, ...).

***ProtoType_PREDICATE*** Prototype used by predicate callbacks (e.g., check_integration_step, ...).

***ProtoType_STATIC_NON_MODIFIER*** Prototype used by static non-modifier callbacks (e.g., static_construct, ...).

***ProtoType_STATIC_PREDICATE*** Prototype used by static predicate callbacks (e.g., static_check_integration_step, ...).

***PROTOTYPES_L*** Number of different prototype types.

***ProtoType_NONE*** Default value for non-initialized ProtoTypes variables.

#### 2.1.3.4 enum SPARK::ReturnTypes

Enum for return types from modifier callbacks.

**Enumeration values:**

***ReturnType_VALUE*** Indicates that scalar value written to the target TVariable object is the variable value.

***ReturnType_RESIDUAL*** Indicates that scalar value written to the target TVariable object is the residual value of the function matched with this variable.

***RETURNTYPES_L*** Number of different return types.

***ReturnType_NONE*** Default value for non-initialized ReturnTypes variables.

#### 2.1.3.5 enum SPARK::VariableTypes

Enum for the various variable types in the problem.

**Enumeration values:**

***VariableType_GLOBAL_TIME*** Refers to the LINK tagged with the GLOBAL_TIME keyword.

***VariableType_GLOBAL_TIME_STEP*** Refers to the LINK tagged with the GLOBAL_TIME_TIME keyword.

***VariableType_PARAMETER*** Refers to the LINKs specified with the PARAMETER keyword.

***VariableType_INPUT*** Refers to the LINKs specified with the INPUT keyword.

***VariableType_UNKNOWN*** Refers to the LINKs specified with none of the above keywords.

***VARIABLETYPES_L*** Number of different variable types.

***VariableType_NONE*** Default value for non-initialized VariableTypes variables.

#### 2.1.3.6 enum SPARK::RequestTypes

Enum for the various atomic class requests

The values are specified by the following bit masks that can be combined using the bitwise OR ( | ) operator.

**Enumeration values:**

***RequestType_ABORT*** Refers to a SPARK::TDispatcher::abort() request (see macro REQUEST__ABORT in spark.h).

***RequestType_STOP*** Refers to a SPARK::TDispatcher::stop() request (see macro REQUEST__STOP in spark.h).

***RequestType_SET_STOP_TIME*** Refers to a SPARK::TDispatcher::set_stop_time() request (see macro REQUEST__SET_-STOP_TIME in spark.h).

***RequestType_REPORT*** Refers to a SPARK::TDispatcher::report() request (see macro REQUEST__REPORT in spark.h).

***RequestType_SNAPSHOT*** Refers to a SPARK::TDispatcher::snapshot() request (see macro REQUEST__SNAPSHOT in spark.h).

***RequestType_SET_MEETING_POINT*** Refers to a SPARK::TDispatcher::set_meeting_point() request (see macro REQUEST__SET_MEETING_POINT in spark.h).

***RequestType_CLEAR_MEETING_POINTS*** Refers to a SPARK::TDispatcher::clear_meeting_points() request (see macro REQUEST__CLEAR_MEETING_POINTS in spark.h).

***RequestType_RESTART*** Refers to a SPARK::TDispatcher::restart() request (see macro REQUEST__RESTART in spark.h).

***RequestType_SET_TIME_STEP*** Refers to a SPARK::TDispatcher::set_time_step() request (see macro REQUEST__SET_-TIME_STEP in spark.h).

***RequestType_SET_DYNAMIC_STEPPER*** Refers to a SPARK::TDispatcher::set_dynamic_stepper() request (see macro REQUEST__SET_DYNAMIC_STEPPER in spark.h).

***REQUESTTYPES_L*** Number of different request types.

***RequestType_NONE*** Default value for an invalid request type.

## 2.1.4 Function Documentation

### 2.1.4.1 std::ostream& operator<< (std::ostream & *os*, const SPARK::TUnknown & *unknown*)

Output operator << overload for the SPARK::TUnknown class.

### 2.1.4.2 std::ostream& operator<< (std::ostream & *os*, const SPARK::TVariable & *V*)

Output operator << overload for the SPARK::TVariable class.

### 2.1.4.3 double abs (double *x*) [inline]

Returns the absolute value of x

**Warning:**
Use SPARK::abs() to avoid possible confusion with abs(), fabs() defined in the system include files <math.h> or . The abs() function has the prototype int abs(int x) which forces a type conversion from double to int. Usually, this is not the desired effect and it is a tricky bug hard to detect. Therefore, we recommend using SPARK::abs() to avoid any confusion.

### 2.1.4.4 double min (double *x*, double *y*) [inline]

Returns the min of two variables that can be compared using <

**Warning:**
Use SPARK::min() to avoid possible confusion with std::min()

### 2.1.4.5 double max (double *x*, double *y*) [inline]

Returns the max of two variables that can be compared using >

**Warning:**
Use SPARK::max() to avoid possible confusion with std::max()

#### 2.1.4.6 double sign (double *x*) [inline]

Returns the +1.0 if x is positive, -1.0 otherwise.

#### 2.1.4.7 double sign (double *retval*, double *test*) [inline]

Returns the a if b is strictly positive, -a otherwise.

#### 2.1.4.8 double log2 (double *x*) [inline]

Returns the log in base 2 of the double x

**Note:**
Use log() and log10() declared in to compute the logarithms in base e and 10 respectively

#### 2.1.4.9 template<typename T> NumericalValueTypes InfiniteOrNaN (T *scalar*)

Checks whether the floating-point number scalar is infinite or NaN or a valid numerical scalar.

**Returns:**
Numerical type as enum NumericalValueTypes

**Parameters:**
***scalar*** floating-point value of type T to check

#### 2.1.4.10 std::ostream& operator<< (std::ostream & *os*, const SPARK::TArgument & *argument*)

Output operator << overload for the TArgument class.

#### 2.1.4.11 std::ostream& operator<< (std::ostream & *os*, const SPARK::TTarget & *target*)

Output operator << overload for the TTarget class.

### 2.1.5 Variable Documentation

#### 2.1.5.1 const double SPARK::TINY = 1.0E-30

Considered essentially as zero in solver (e.g., used to detect singularity).

#### 2.1.5.2 const double SPARK::SQRT_UROUND = sqrt(UROUND)

Square root of the unit round-off error.

## 2.2 SPARK::AtomicClass Namespace Reference

Definition of the atomic class API library used to manage the private data from the callbacks asociated with each SPARK::TObject and SPARK::TInverse instances.

### Functions

- template<typename DataType> void SetData (TObject *object, DataType *data)

  *Sets pointer to private data in TObject instance.*

- template<typename StaticDataType> void SetData (TInverse *inverse, StaticDataType *staticData)

  *Sets pointer to static private data in TInverse instance.*

- template<typename DataType> DataType * GetData (TObject *object)

  *Returns address of private data of type* `DataType` *attached to the* `TObject` *instance.*

- template<typename StaticDataType> StaticDataType * GetData (TInverse *inverse)

  *Returns address of static private data of type* `StaticDataType` *attached to the* `TInverse` *instance.*

- template<typename DataType, typename DeletePolicyType> bool DeleteData (TObject *object, DataType *dummy1=0, DeletePolicyType *dummy2=0)

  *Deallocates private data of type* `DataType` *attached to a TObject instance.*

- template<typename StaticDataType, typename DeletePolicyType> bool DeleteData (TInverse *inverse, StaticDataType *dummy1=0, DeletePolicyType *dummy2=0)

  *Deallocates private data of type* `StaticDataType` *attached to a TInverse instance.*

### 2.2.1 Detailed Description

Definition of the atomic class API library used to manage the private data from the callbacks asociated with each SPARK::TObject and SPARK::TInverse instances.

### 2.2.2 Function Documentation

#### 2.2.2.1 template<typename DataType> void SetData (TObject * *object*, DataType * *data*)

Sets pointer to private data in TObject instance.

**Parameters:**

***object*** Address of the TObject instance (use THIS macro from within the callback body)

***data*** Address of the private data to be stored within the TObject instance

Here is the call graph for this function:

#### 2.2.2.2 template<typename StaticDataType> void SetData (TInverse * *inverse*, StaticDataType * *staticData*)

Sets pointer to static private data in TInverse instance.

**Parameters:**

***inverse*** Address of the TInverse instance (use THIS macro from within the static callback body)

***staticData*** Address of the static private data to be stored within the TInverse instance

Here is the call graph for this function:


#### 2.2.2.3 template<typename DataType> DataType* GetData (TObject * *object*)

Returns address of private data of type `DataType` attached to the `TObject` instance.

**Returns:**

Pointer to private data as DataType*

**Parameters:**

***object*** (use THIS macro from within the callback body)

Here is the call graph for this function:


#### 2.2.2.4 template<typename StaticDataType> StaticDataType* GetData (TInverse * *inverse*)

Returns address of static private data of type `StaticDataType` attached to the `TInverse` instance.

**Returns:**

Pointer to static private data as StaticDataType*

**Parameters:**

***inverse*** (use THIS macro from within the static callback body)

Here is the call graph for this function:


#### 2.2.2.5 template<typename DataType, typename DeletePolicyType> bool DeleteData (TObject * *object*, DataType * *dummy1* = 0, DeletePolicyType * *dummy2* = 0)

Deallocates private data of type `DataType` attached to a TObject instance.

**Parameters:**

***object*** Address of the TObject instance (use THIS macro from within the callback body)

***dummy1*** Never specify this parameter. It is used as a workaround for a bug in MSVC++

***dummy2*** Never specify this parameter. It is used as a workaround for a bug in MSVC++

**Precondition:**
DeletePolicyType must define a static method named apply() that performs the deletion task. Possible policy candidates are: SPARK::delete_policy and SPARK::delete_array_policy

**Postcondition:**
Private data is deallocated and pointer in TObject instance is reset to 0

#### 2.2.2.6 template<typename StaticDataType, typename DeletePolicyType> bool DeleteData (TInverse * *inverse*, StaticDataType * *dummy1* = 0, DeletePolicyType * *dummy2* = 0)

Deallocates private data of type `StaticDataType` attached to a TInverse instance.

**Parameters:**
***inverse*** Address of the TInverse instance (use THIS macro from within the static callback body)

***dummy1*** Never specify this parameter. It is used as a workaround for a bug in MSVC++

***dummy2*** Never specify this parameter. It is used as a workaround for a bug in MSVC++

**Precondition:**
DeletePolicyType must define a static method named apply() that performs the deletion task. Possible policy candidates are: SPARK::delete_policy and SPARK::delete_array_policy

**Postcondition:**
Private data is deallocated and pointer in TInverse instance is reset to 0

## 2.3 SPARK::Requests Namespace Reference

Declarations of classes and functions needed to send/dispatch/process the SPARK requests.

### Classes

- class THeader

  *Describes where the request is sent from and the target problem. Used by the TDispatcher class to dispatch requests.*

- class TDispatcher

  *Wrapper class that defines a static function for each request that can be sent from an atomic class. Use in coordination with the class THeader for the request addressing.*

### 2.3.1 Detailed Description

Declarations of classes and functions needed to send/dispatch/process the SPARK requests.

# 2.4 SPARK::Variable Namespace Reference

Definition of classes and free functions used to implement the SPARK::TVariable class and the various interfaces to a SPARK::TVariable object.

## Classes

- class TInterface

  *Base class for all interfaces to a SPARK::TVariable object. Read and write methods for the current values are provided as protected methods to can be used in the derived classes to implement the proper behavior for each type of interface.*

## Functions

- double ComputeScale (double tol, double atol, double value)

  *Computes scale for a problem variable using the relative tolerance* `tol` *and the absolute tolerance* `atol`.

- void Write (std::ostream &os, const SPARK::TVariable &variable)

  *Writes out to the stream os the name and unit of the SPARK::TVariable object.*

- void WriteMatchedObject (std::ostream &os, const SPARK::TVariable &variable, const std::string &before)

  *Write argument list of evaluate callback for object associated with variable if any.*

### 2.4.1 Detailed Description

Definition of classes and free functions used to implement the SPARK::TVariable class and the various interfaces to a SPARK::TVariable object.

### 2.4.2 Function Documentation

#### 2.4.2.1 double ComputeScale (double *tol*, double *atol*, double *value*)

Computes scale for a problem variable using the relative tolerance `tol` and the absolute tolerance `atol`.

**Returns:**
scale as double

**Parameters:**
***tol*** relative tolerance where log(tol) = -(number of significant digits)
***atol*** absolute tolerance for this variable (essentially the value we consider to be zero)
***value*** current value of this variable

#### 2.4.2.2 void Write (std::ostream & *os*, const SPARK::TVariable & *variable*)

Writes out to the stream os the name and unit of the SPARK::TVariable object.

#### 2.4.2.3 void WriteMatchedObject (std::ostream & *os*, const SPARK::TVariable & *variable*, const std::string & *before*)

Write argument list of evaluate callback for object associated with variable if any.

# Chapter 3

# SPARK Atomic Class API Class Documentation

## 3.1 SPARK::delete_array_policy Struct Reference

Policy class used to delete array data types using delete [] operator.

`#include <classapi.h>`

### 3.1.1 Detailed Description

Policy class used to delete array data types using delete [] operator.

**Note:**
The static method apply() does not throw any exceptions.

The documentation for this struct was generated from the following file:

- classapi.h

## 3.2 SPARK::delete_policy Struct Reference

Policy class used to delete non-array data types using delete operator.

`#include <classapi.h>`

### 3.2.1 Detailed Description

Policy class used to delete non-array data types using delete operator.

**Note:**
The static method apply() does not throw any exceptions.

The documentation for this struct was generated from the following file:

- classapi.h

## 3.3 SPARK::deleter< IsArray > Struct Template Reference

Helper class that implements either delete_policy or delete_array_policy depending on the compile-time value of the non-type parameter IsArray.

`#include <classapi.h>`

### 3.3.1 Detailed Description

**template<bool IsArray> struct SPARK::deleter< IsArray >**

Helper class that implements either delete_policy or delete_array_policy depending on the compile-time value of the non-type parameter IsArray.

The delete policy is used in the SPARK::AtomicClass::DeleteData() API function. Also see macro DELETE_DATA in file spark.h for usage of the deleter<> struct.

The documentation for this struct was generated from the following file:

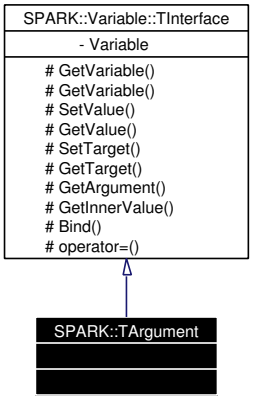
- classapi.h

## 3.4 SPARK::TArgument Class Reference

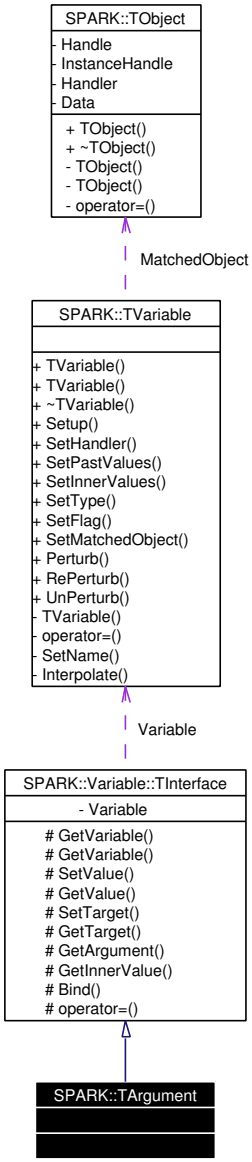
This class acts as a read-only interface to a TVariable object. It is used only in the callbacks to describe the argument variables. This class is not used internally in the solver. It provides proper value access behavior depending on the type of the variable and the calling context. Also, it implements some of the attribute access methods from TVariable.

`#include <callback.h>`

Inheritance diagram for SPARK::TArgument:




Collaboration diagram for SPARK::TArgument:




## Public Member Functions

### Structors

- TArgument ()
  *Default constructor.*
- TArgument (SPARK::TVariable *variable)
  *Normal constructor using a pointer to a TVariable object.*
- TArgument (const TArgument &argument)
  *Copy constructor.*

### Numerical access methods

- double operator() (unsigned innerStep) const

*Returns the inner value for innerStep.*

- operator double () const
  *Allows implicit conversion to double by returning the current value.*

### 3.4.1 Detailed Description

This class acts as a read-only interface to a TVariable object. It is used only in the callbacks to describe the argument variables. This class is not used internally in the solver. It provides proper value access behavior depending on the type of the variable and the calling context. Also, it implements some of the attribute access methods from TVariable.

### 3.4.2 Constructor & Destructor Documentation

#### 3.4.2.1 SPARK::TArgument::TArgument () [inline]

Default constructor.

#### 3.4.2.2 SPARK::TArgument::TArgument (SPARK::TVariable * *variable*) [inline, explicit]

Normal constructor using a pointer to a TVariable object.

#### 3.4.2.3 SPARK::TArgument::TArgument (const TArgument & *argument*) [inline]

Copy constructor.

### 3.4.3 Member Function Documentation

#### 3.4.3.1 double SPARK::TArgument::operator() (unsigned *innerStep*) const [inline]

Returns the inner value for innerStep.

#### 3.4.3.2 SPARK::TArgument::operator double () const [inline]

Allows implicit conversion to double by returning the current value.

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

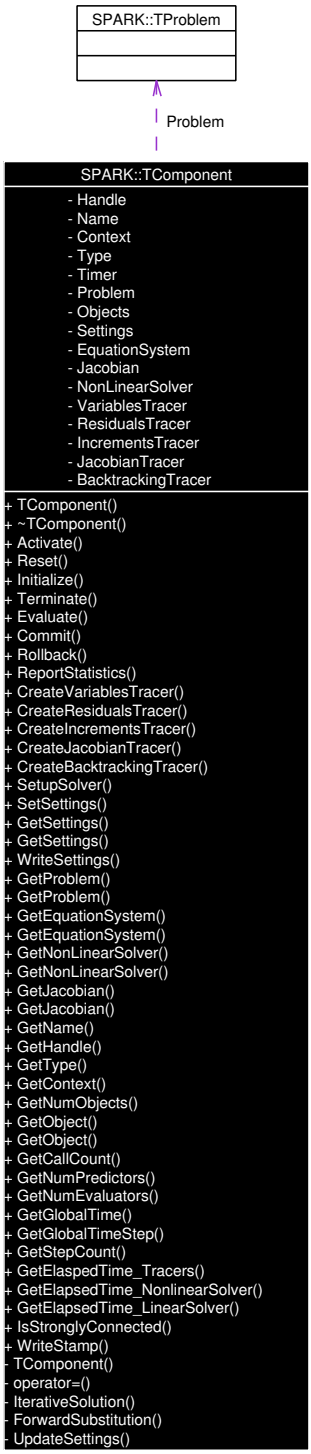
- callback.h

## 3.5 SPARK::TComponent Class Reference

Class that solves the set of DAE equations generated by setupcpp.

`#include <component.h>`

Collaboration diagram for SPARK::TComponent:



## Public Types

- enum ComponentTypes {

  ComponentType_WEAK = 0,

  ComponentType_STRONG = 1 }

  *Type to indicate whether a component consists of a set of weakly- or strongly-connected equations.*

## Public Member Functions

- TComponent (unsigned handle, unsigned numNormalUnknowns, SPARK::TUnknown normalUnknowns[ ], unsigned numBreakUnknowns, SPARK::TUnknown breakUnknowns[ ], unsigned numObjects, SPARK::TObject ∗objects[ ]) throw (SPARK::XMemory)

  *Constructor.*

- ∼TComponent () throw ()

  *Destructor.*

- void Evaluate ()

  *Solves the set of differential-algebraic equations.*

- const char ∗ GetName () const

  *Returns the name of the component as const char∗.*

- unsigned GetHandle () const

  *Returns the unique handle assigne to this component as an unsigned int.*

- unsigned GetType () const

  *Returns the component type as unsigned (see enum TComponent::TType).*

- unsigned GetNumObjects () const

  *Returns the number of objects comprising this component.*

- SPARK::TObject ∗ GetObject (unsigned i)

  *Returns a pointer to the TObject object evaluated in position i.*

- const SPARK::TObject ∗ GetObject (unsigned i) const

  *Returns a const pointer to the TObject object evaluated in position i.*

- bool IsStronglyConnected () const

  *Returns true if this component is a strongly-connected component.*

### 3.5.1 Detailed Description

Class that solves the set of DAE equations generated by setupcpp.

A problem consists of a series of TComponent objects that are solved in a fixed order, from the first component (with index 0) to the last component. The order in which the components need to be solved reflects the topological dependencies derived by setupcpp.

## 3.5.2 Member Enumeration Documentation

### 3.5.2.1 enum SPARK::TComponent::ComponentTypes

Type to indicate whether a component consists of a set of weakly- or strongly-connected equations.

**Enumeration values:**

***ComponentType_WEAK*** Indicates a weakly-connected component that is solved using forward substitution.

***ComponentType_STRONG*** Indicates a strongly-connected component that requires iterative solution.

## 3.5.3 Constructor & Destructor Documentation

### 3.5.3.1 SPARK::TComponent::TComponent (unsigned *handle*, unsigned *numNormalUnknowns*, SPARK::TUnknown *normalUnknowns*[ ], unsigned *numBreakUnknowns*, SPARK::TUnknown *breakUnknowns*[ ], unsigned *numObjects*, SPARK::TObject * *objects*[ ]) throw (SPARK::XMemory)

Constructor.

### 3.5.3.2 SPARK::TComponent::~TComponent () throw ()

Destructor.

## 3.5.4 Member Function Documentation

### 3.5.4.1 void SPARK::TComponent::Evaluate ()

Solves the set of differential-algebraic equations.

**Exceptions:**

***SPARK::XNoConvergence*** Thrown if convergence cannot be obtained.

***SPARK::XBadNumerics*** Thrown if bad numerics detected.

### 3.5.4.2 const char* SPARK::TComponent::GetName () const [inline]

Returns the name of the component as const char*.

### 3.5.4.3 unsigned SPARK::TComponent::GetHandle () const [inline]

Returns the unique handle assigne to this component as an unsigned int.

### 3.5.4.4 unsigned SPARK::TComponent::GetType () const [inline]

Returns the component type as unsigned (see enum TComponent::TType).

### 3.5.4.5 unsigned SPARK::TComponent::GetNumObjects () const [inline]

Returns the number of objects comprising this component.

#### 3.5.4.6 SPARK::TObject* SPARK::TComponent::GetObject (unsigned *i*) [inline]

Returns a pointer to the TObject object evaluated in position i.

**Note:**
The first object evaluated in this component is stored in position 0.

#### 3.5.4.7 const SPARK::TObject* SPARK::TComponent::GetObject (unsigned *i*) const [inline]

Returns a const pointer to the TObject object evaluated in position i.

**Note:**
The first object evaluated in this component is stored in position 0.

#### 3.5.4.8 bool SPARK::TComponent::IsStronglyConnected () const [inline]

Returns true if this component is a strongly-connected component.

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

- component.h

## 3.6 SPARK::Requests::TDispatcher Class Reference

Wrapper class that defines a static function for each request that can be sent from an atomic class. Use in coordination with the class THeader for the request addressing.

`#include <requests.h>`

### Static Public Member Functions

#### Requests processed when detected

- bool abort (const THeader &header)
  *Aborts the simulation by exiting "cleanly" from the process.*

#### Requests processed only after a successful step

- bool stop (const THeader &header)
  *Stops the simulation after the next successful step.*
- bool set_stop_time (const THeader &header, double time)
  *Sets the time* `time` *at which the simulation will be stopped.*
- bool report (const THeader &header)
  *Generates a "forced" report at the current time.*
- bool snapshot (const THeader &header, const char *filename)
  *Generates a snapshot file named filename where filename can include a valid path.*
- bool set_meeting_point (const THeader &header, double time)
  *Forces the time stepper to synchronize with time, whereby time > current time.*
- bool clear_meeting_points (const THeader &header)
  *Clears all meeting points assumulated so far through synchronize requests.*
- bool restart (const THeader &header)
  *Restarts the simulation with a static step in order to perform a consistent initialization calculation.*

#### Restricted requests

- bool set_time_step (const THeader &header, double h)
  *Sets the time step h for the next dynamic step.*
- bool set_dynamic_stepper (const THeader &header)
  *Sets dynamic stepper that will be used from now on for all dynamic steps.*

### 3.6.1 Detailed Description

Wrapper class that defines a static function for each request that can be sent from an atomic class. Use in coordination with the class THeader for the request addressing.

**Note:**
The dispatch functions return true if the dispatcher was capable to pass the request to the desired target problem. Otherwise, the functions return false to indicate that the request will not be taken into account.

## 3.6.2 Member Function Documentation

### 3.6.2.1 bool SPARK::Requests::TDispatcher::abort (const THeader & *header*) [static]

Aborts the simulation by exiting "cleanly" from the process.

**Parameters:**

***header*** Request header

### 3.6.2.2 bool SPARK::Requests::TDispatcher::stop (const THeader & *header*) [static]

Stops the simulation after the next successful step.

**Parameters:**

***header*** Request header

### 3.6.2.3 bool SPARK::Requests::TDispatcher::set_stop_time (const THeader & *header*, double *time*) [static]

Sets the time `time` at which the simulation will be stopped.

This request is accepted only if the requested stop time is ahead of the current global time of the simulator. It will however overwrite a previous stop time request if the new one indicates an earlier stopping time. The stop time does not overwrite the final time specified in the runtime control parameters.

**Parameters:**

***header*** Request header

***time*** Stopping time

### 3.6.2.4 bool SPARK::Requests::TDispatcher::report (const THeader & *header*) [static]

Generates a "forced" report at the current time.

**Parameters:**

***header*** Request header

### 3.6.2.5 bool SPARK::Requests::TDispatcher::snapshot (const THeader & *header*, const char * *filename*) [static]

Generates a snapshot file named filename where filename can include a valid path.

**Parameters:**

***header*** Request header

***filename*** Name of the snapshot file

### 3.6.2.6 bool SPARK::Requests::TDispatcher::set_meeting_point (const THeader & *header*, double *time*) [static]

Forces the time stepper to synchronize with time, whereby time $>$ current time.

**Parameters:**

***header*** Request header

***time*** Meeting point

#### 3.6.2.7 bool SPARK::Requests::TDispatcher::clear_meeting_points (const THeader & *header*) [static]

Clears all meeting points assumulated so far through synchronize requests.

**Parameters:**

***header*** Request header

#### 3.6.2.8 bool SPARK::Requests::TDispatcher::restart (const THeader & *header*) [static]

Restarts the simulation with a static step in order to perform a consistent initialization calculation.

**Parameters:**

***header*** Request header

#### 3.6.2.9 bool SPARK::Requests::TDispatcher::set_time_step (const THeader & *header*, double *h*) [static]

Sets the time step h for the next dynamic step.

**Parameters:**

***header*** Request header

***h*** Candidate time step

#### 3.6.2.10 bool SPARK::Requests::TDispatcher::set_dynamic_stepper (const THeader & *header*) [static]

Sets dynamic stepper that will be used from now on for all dynamic steps.

**Parameters:**

***header*** Request header

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

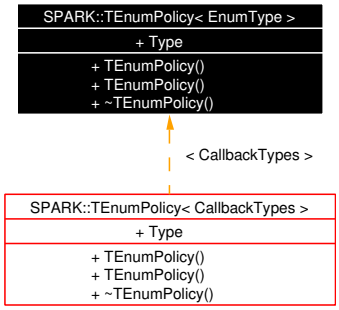
- requests.h

## 3.7 SPARK::TEnumPolicy< EnumType > Struct Template Reference

Policy class that specifies a enum value of type EnumType.

`#include <callback.h>`

Inheritance diagram for SPARK::TEnumPolicy< EnumType >:




### 3.7.1 Detailed Description

**template<typename EnumType> struct SPARK::TEnumPolicy< EnumType >**

Policy class that specifies a enum value of type EnumType.

The documentation for this struct was generated from the following file:

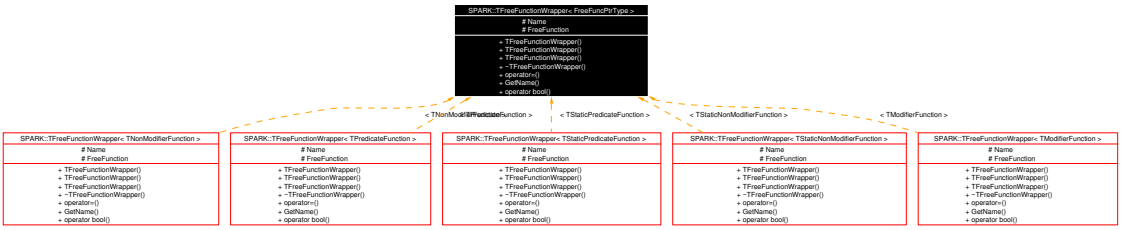
- callback.h

# 3.8 SPARK::TFreeFunctionWrapper< FreeFuncPtrType > Class Template Reference

Wrapper class for any free function that is invoked as an atomic class callback.

`#include <callback.h>`

Inheritance diagram for SPARK::TFreeFunctionWrapper< FreeFuncPtrType >:




## Public Member Functions

- TFreeFunctionWrapper ()

  *Default constructor.*

- TFreeFunctionWrapper (const TFreeFunctionWrapper &w)

  *Copy constructor.*

- TFreeFunctionWrapper (const char *name, FreeFuncPtrType t)

  *Constructor.*

- ~TFreeFunctionWrapper () throw ()

  *Destructor.*

- TFreeFunctionWrapper & operator= (const TFreeFunctionWrapper &w)

  *Assignment operator.*

- const char * GetName () const

  *Returns name of the callback function as const char*.*

- operator bool () const

  *Returns true if wrapper contains a valid callback function.*

### 3.8.1 Detailed Description

**template<typename FreeFuncPtrType> class SPARK::TFreeFunctionWrapper< FreeFuncPtrType >**

Wrapper class for any free function that is invoked as an atomic class callback.

### 3.8.2 Constructor & Destructor Documentation

#### 3.8.2.1 template<typename FreeFuncPtrType> SPARK::TFreeFunctionWrapper< FreeFuncPtrType >::TFreeFunctionWrapper () [inline]

Default constructor.

#### 3.8.2.2 template<typename FreeFuncPtrType> SPARK::TFreeFunctionWrapper< FreeFuncPtrType >::TFreeFunctionWrapper (const TFreeFunctionWrapper< FreeFuncPtrType > & *w*) [inline]

Copy constructor.

#### 3.8.2.3 template<typename FreeFuncPtrType> SPARK::TFreeFunctionWrapper< FreeFuncPtrType >::TFreeFunctionWrapper (const char * *name*, FreeFuncPtrType *t*) [inline]

Constructor.

#### 3.8.2.4 template<typename FreeFuncPtrType> SPARK::TFreeFunctionWrapper< FreeFuncPtrType >::~TFreeFunctionWrapper () throw () [inline]

Destructor.

### 3.8.3 Member Function Documentation

#### 3.8.3.1 template<typename FreeFuncPtrType> TFreeFunctionWrapper& SPARK::TFreeFunctionWrapper< FreeFuncPtrType >::operator= (const TFreeFunctionWrapper< FreeFuncPtrType > & *w*) [inline]

Assignment operator.

#### 3.8.3.2 template<typename FreeFuncPtrType> const char* SPARK::TFreeFunctionWrapper< FreeFuncPtrType >::GetName () const [inline]

Returns name of the callback function as const char*.

#### 3.8.3.3 template<typename FreeFuncPtrType> SPARK::TFreeFunctionWrapper< FreeFuncPtrType >::operator bool () const [inline]

Returns true if wrapper contains a valid callback function.

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

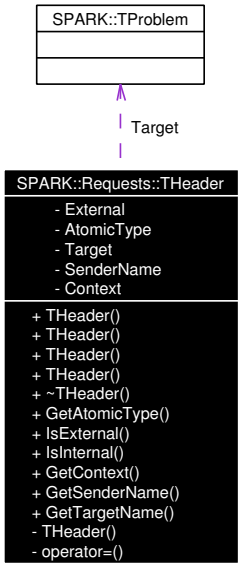
- callback.h

# 3.9 SPARK::Requests::THeader Class Reference

Describes where the request is sent from and the target problem. Used by the TDispatcher class to dispatch requests.

`#include <requests.h>`

Collaboration diagram for SPARK::Requests::THeader:




## Public Member Functions

- THeader (const TObject *from, const TProblem *to, const char *context)

  *Constructs header for request sent by TObject object.*

- THeader (const TInverse *from, const TProblem *to, const char *context)

  *Constructs header for request sent by TInverse object.*

- THeader (const TProblem *from, const TProblem *to, const char *context)

  *Constructs header for request sent by TProblem object.*

- THeader (const THeader &header)

  *Copy constructor.*

- ~THeader () throw ()

  *Trivial destructor.*

- SPARK::AtomicTypes GetAtomicType () const

  *Returns atomic type of sender object.*

- bool IsExternal () const

  *Returns true if external request.*

- bool IsInternal () const

  *Returns true if internal request.*

- const char * GetContext () const

  *Returns context as const char*.*

- const char * GetSenderName () const

  *Returns name of sender object as const char*.*

- const char * GetTargetName () const

  *Returns name of target problem as const char*.*

### 3.9.1 Detailed Description

Describes where the request is sent from and the target problem. Used by the TDispatcher class to dispatch requests.

### 3.9.2 Constructor & Destructor Documentation

#### 3.9.2.1 SPARK::Requests::THeader::THeader (const TObject * *from*, const TProblem * *to*, const char * *context*)

Constructs header for request sent by TObject object.

#### 3.9.2.2 SPARK::Requests::THeader::THeader (const TInverse * *from*, const TProblem * *to*, const char * *context*)

Constructs header for request sent by TInverse object.

#### 3.9.2.3 SPARK::Requests::THeader::THeader (const TProblem * *from*, const TProblem * *to*, const char * *context*)

Constructs header for request sent by TProblem object.

#### 3.9.2.4 SPARK::Requests::THeader::THeader (const THeader & *header*)

Copy constructor.

#### 3.9.2.5 SPARK::Requests::THeader::~THeader () throw () [inline]

Trivial destructor.

### 3.9.3 Member Function Documentation

#### 3.9.3.1 SPARK::AtomicTypes SPARK::Requests::THeader::GetAtomicType () const [inline]

Returns atomic type of sender object.

#### 3.9.3.2 bool SPARK::Requests::THeader::IsExternal () const [inline]

Returns true if external request.

#### 3.9.3.3 bool SPARK::Requests::THeader::IsInternal () const [inline]

Returns true if internal request.

#### 3.9.3.4 const char* SPARK::Requests::THeader::GetContext () const [inline]

Returns context as const char*.

#### 3.9.3.5 const char* SPARK::Requests::THeader::GetSenderName () const [inline]

Returns name of sender object as const char*.

#### 3.9.3.6 const char* SPARK::Requests::THeader::GetTargetName () const

Returns name of target problem as const char*.

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

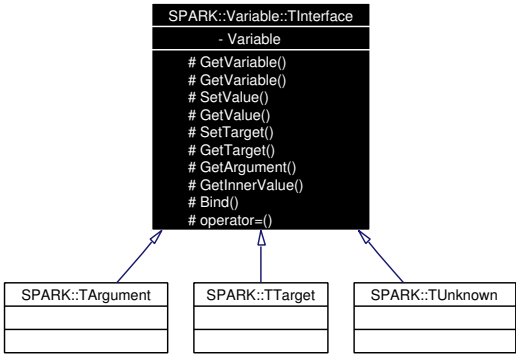
- requests.h

## 3.10 SPARK::Variable::TInterface Class Reference

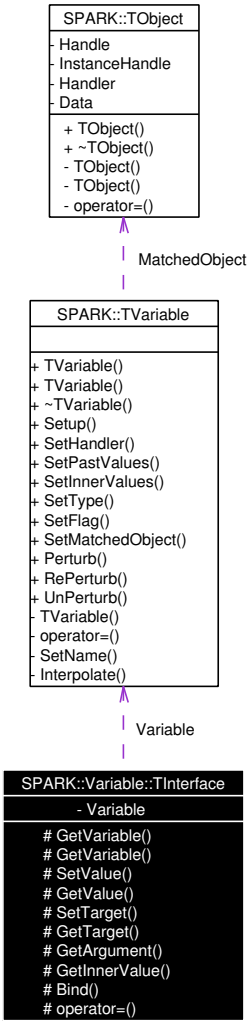
Base class for all interfaces to a SPARK::TVariable object. Read and write methods for the current values are provided as protected methods to can be used in the derived classes to implement the proper behavior for each type of interface.

`#include <variable.h>`

Inheritance diagram for SPARK::Variable::TInterface:




Collaboration diagram for SPARK::Variable::TInterface:




## Public Member Functions

### Structors

- TInterface ()
  *Default constructor.*
- TInterface (SPARK::TVariable *variable)
  *Normal constructor using a pointer to a SPARK::TVariable object.*
- TInterface (const TInterface &interf)
  *Copy constructor.*
- virtual ~TInterface () throw ()
  *Destructor.*

### Restricted access methods for variable properties

- unsigned GetHandle () const
  *Returns the unique handle as unsigned.*

- SPARK::VariableTypes GetType () const
  *Returns the variable type as SPARK::VariableTypes.*
- unsigned GetFlag () const
  *Returns internal flag.*
- double GetPastValue (unsigned idx) const
  *Returns the past value from idx steps ago.*
- double operator[ ] (unsigned idx) const
  *Returns the past value from idx steps ago.*
- const std::string & GetName () const
  *Returns the variable name of the underlying SPARK::TVariable object as const std::string&.*
- const std::string & GetUnit () const
  *Returns the variable unit string of the underlying SPARK::TVariable object as const std::string&.*
- void SetUnit (const char *unit)
  *Sets the unit string for the underlying variable.*
- double GetInit () const
  *Returns the inital value of the underlying SPARK::TVariable object as double.*
- void SetInit (double init)
  *Sets the initial value to init.*
- double GetMin () const
  *Returns the minimum value of the underlying SPARK::TVariable object as double.*
- void SetMin (double min)
  *Sets the minimum value to min.*
- double GetMax () const
  *Returns the maximum value of the underlying SPARK::TVariable object as double.*
- void SetMax (double max)
  *Sets the maximum value to max.*
- double GetAbsTolerance () const
  *Returns the absolute tolerance of the underlying SPARK::TVariable object as double.*
- void SetAbsTolerance (double atol)
  *Sets the absolute tolerance to atol.*
- double GetScale () const
  *Returns the current scale.*
- const char * GetReadUrlString () const
  *Returns Read URL string.*
- int GetReadUrlHandle () const
  *Returns Read URL handle.*
- const char * GetWriteUrlString () const
  *Returns Write URL string.*

- int GetWriteUrlHandle () const
  *Returns Write URL handle.*

**Predicate methods**

**IO methods**

- void Write (std::ostream &os) const
  *Writes the variable name, unit and current value to the output stream os.*

## Protected Member Functions

- SPARK::TVariable * GetVariable ()
  *Returns underlying pointer.*
- void SetValue (double scalar)
  *Sets the current value to scalar.*
- double GetValue () const
  *Allows implicit conversion to double by returning the current value.*
- void SetTarget (double scalar)
  *Sets the current target value to scalar.*
- double GetTarget () const
  *Returns value from evaluating the matched object (used by TEquationSystem and TJacobian internally).*
- double GetArgument () const
  *Returns the current argument value as a double.*
- double GetInnerValue (unsigned innerStep) const
  *Returns the inner value for innerStep.*
- void Bind (SPARK::TVariable *variable)
  *Used by derived classes if required (see TUnknown).*

### 3.10.1 Detailed Description

Base class for all interfaces to a SPARK::TVariable object. Read and write methods for the current values are provided as protected methods to can be used in the derived classes to implement the proper behavior for each type of interface.

### 3.10.2 Constructor & Destructor Documentation

#### 3.10.2.1 SPARK::Variable::TInterface::TInterface ()

Default constructor.

#### 3.10.2.2 SPARK::Variable::TInterface::TInterface (SPARK::TVariable * *variable*) [explicit]

Normal constructor using a pointer to a SPARK::TVariable object.

#### 3.10.2.3 SPARK::Variable::TInterface::TInterface (const TInterface & *interf*)

Copy constructor.

#### 3.10.2.4 virtual SPARK::Variable::TInterface::~TInterface () throw () [inline, virtual]

Destructor.

### 3.10.3 Member Function Documentation

#### 3.10.3.1 unsigned SPARK::Variable::TInterface::GetHandle () const

Returns the unique handle as unsigned.

#### 3.10.3.2 SPARK::VariableTypes SPARK::Variable::TInterface::GetType () const

Returns the variable type as SPARK::VariableTypes.

#### 3.10.3.3 unsigned SPARK::Variable::TInterface::GetFlag () const

Returns internal flag.

#### 3.10.3.4 double SPARK::Variable::TInterface::GetPastValue (unsigned *idx*) const

Returns the past value from idx steps ago.

#### 3.10.3.5 ]

double SPARK::Variable::TInterface::operator[ ] (unsigned *idx*) const [inline]

Returns the past value from idx steps ago.

#### 3.10.3.6 const std::string& SPARK::Variable::TInterface::GetName () const

Returns the variable name of the underlying SPARK::TVariable object as const std::string&.

#### 3.10.3.7 const std::string& SPARK::Variable::TInterface::GetUnit () const

Returns the variable unit string of the underlying SPARK::TVariable object as const std::string&.

#### 3.10.3.8 void SPARK::Variable::TInterface::SetUnit (const char * *unit*)

Sets the unit string for the underlying variable.

#### 3.10.3.9 double SPARK::Variable::TInterface::GetInit () const

Returns the inital value of the underlying SPARK::TVariable object as double.

#### 3.10.3.10 void SPARK::Variable::TInterface::SetInit (double *init*)

Sets the initial value to init.

#### 3.10.3.11 double SPARK::Variable::TInterface::GetMin () const

Returns the minimum value of the underlying SPARK::TVariable object as double.

#### 3.10.3.12 void SPARK::Variable::TInterface::SetMin (double *min*)

Sets the minimum value to min.

#### 3.10.3.13 double SPARK::Variable::TInterface::GetMax () const

Returns the maximum value of the underlying SPARK::TVariable object as double.

#### 3.10.3.14 void SPARK::Variable::TInterface::SetMax (double *max*)

Sets the maximum value to max.

#### 3.10.3.15 double SPARK::Variable::TInterface::GetAbsTolerance () const

Returns the absolute tolerance of the underlying SPARK::TVariable object as double.

#### 3.10.3.16 void SPARK::Variable::TInterface::SetAbsTolerance (double *atol*)

Sets the absolute tolerance to atol.

#### 3.10.3.17 double SPARK::Variable::TInterface::GetScale () const

Returns the current scale.

#### 3.10.3.18 const char* SPARK::Variable::TInterface::GetReadUrlString () const

Returns Read URL string.

#### 3.10.3.19 int SPARK::Variable::TInterface::GetReadUrlHandle () const

Returns Read URL handle.

#### 3.10.3.20 const char* SPARK::Variable::TInterface::GetWriteUrlString () const

Returns Write URL string.

#### 3.10.3.21 int SPARK::Variable::TInterface::GetWriteUrlHandle () const

Returns Write URL handle.

#### 3.10.3.22 void SPARK::Variable::TInterface::Write (std::ostream & *os*) const

Writes the variable name, unit and current value to the output stream os.

#### 3.10.3.23 SPARK::TVariable* SPARK::Variable::TInterface::GetVariable () [inline, protected]

Returns underlying pointer.

#### 3.10.3.24 void SPARK::Variable::TInterface::SetValue (double *scalar*) [protected]

Sets the current value to scalar.

#### 3.10.3.25 double SPARK::Variable::TInterface::GetValue () const [protected]

Allows implicit conversion to double by returning the current value.

#### 3.10.3.26 void SPARK::Variable::TInterface::SetTarget (double *scalar*) [protected]

Sets the current target value to scalar.

#### 3.10.3.27 double SPARK::Variable::TInterface::GetTarget () const [protected]

Returns value from evaluating the matched object (used by TEquationSystem and TJacobian internally).

Reimplemented in SPARK::TUnknown.

#### 3.10.3.28 double SPARK::Variable::TInterface::GetArgument () const [protected]

Returns the current argument value as a double.

#### 3.10.3.29 double SPARK::Variable::TInterface::GetInnerValue (unsigned *innerStep*) const [protected]

Returns the inner value for innerStep.

#### 3.10.3.30 void SPARK::Variable::TInterface::Bind (SPARK::TVariable * *variable*) [protected]

Used by derived classes if required (see TUnknown).

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

- variable.h

## 3.11 SPARK::TInverse Class Reference

Class that defines the callbacks for an inverse.

`#include <inverse.h>`

### Public Member Functions

#### Access methods

- unsigned GetHandle () const
  *Returns unique inverse handle as specified in XML file.*
- const char * GetName () const
  *Returns name of inverse as specified in XML file as* `const char*` .
- const char * GetAtomicClassName () const
  *Returns class name as specified in XML file as* `const char*` .
- const char * GetFileName () const
  *Returns file name of atomic class where the inverse is declared as* `const char*` .
- SPARK::AtomicTypes GetAtomicType () const
  *Returns the type of the inverse as SPARK::AtomicTypes.*
- const char * GetActiveCallbackName () const
  *Returns the name of the currently active callback, Returns "" if none active!*
- unsigned GetNumObjects () const
  *Returns the number of TObject instances of this inverse.*
- SPARK::TObject * GetObject (unsigned idx)
  *Returns instance idx as a SPARK::TObject pointer.*
- const SPARK::TProblem * GetProblem () const
  *Returns pointer to SPARK::TProblem object this inverse belongs to.*
- double GetTolerance () const
  *Returns value of relative tolerance currently used by solver.*

#### Data methods

- void * GetStaticData ()
  *Returns static private data as* `void*` .
- void SetStaticData (void *address)
  *Sets the static private data to* `address`.

### 3.11.1 Detailed Description

Class that defines the callbacks for an inverse.

Also contains all instances as SPARK::TObject objects of an inverse, as well as the void* pointer to the static data.

## 3.11.2 Member Function Documentation

### 3.11.2.1 unsigned SPARK::TInverse::GetHandle () const

Returns unique inverse handle as specified in XML file.

### 3.11.2.2 const char* SPARK::TInverse::GetName () const

Returns name of inverse as specified in XML file as `const char*` .

### 3.11.2.3 const char* SPARK::TInverse::GetAtomicClassName () const

Returns class name as specified in XML file as `const char*` .

### 3.11.2.4 const char* SPARK::TInverse::GetFileName () const

Returns file name of atomic class where the inverse is declared as `const char*` .

### 3.11.2.5 SPARK::AtomicTypes SPARK::TInverse::GetAtomicType () const

Returns the type of the inverse as SPARK::AtomicTypes.

### 3.11.2.6 const char* SPARK::TInverse::GetActiveCallbackName () const

Returns the name of the currently active callback, Returns "" if none active!

### 3.11.2.7 unsigned SPARK::TInverse::GetNumObjects () const

Returns the number of TObject instances of this inverse.

### 3.11.2.8 SPARK::TObject* SPARK::TInverse::GetObject (unsigned *idx*)

Returns instance idx as a SPARK::TObject pointer.

### 3.11.2.9 const SPARK::TProblem* SPARK::TInverse::GetProblem () const

Returns pointer to SPARK::TProblem object this inverse belongs to.

### 3.11.2.10 double SPARK::TInverse::GetTolerance () const

Returns value of relative tolerance currently used by solver.

### 3.11.2.11 void* SPARK::TInverse::GetStaticData () [inline]

Returns static private data as `void*` .

#### 3.11.2.12 void SPARK::TInverse::SetStaticData (void * *address*) [inline]

Sets the static private data to `address`.

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

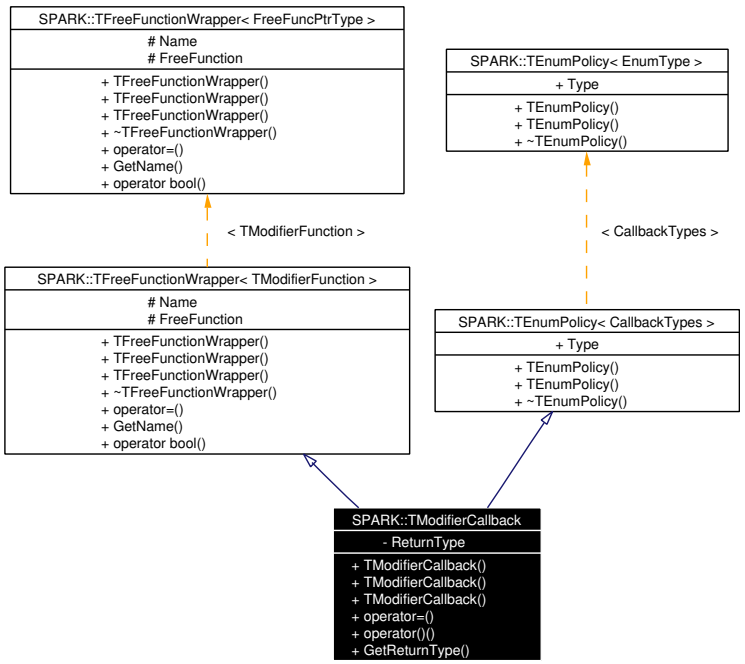
- inverse.h

## 3.12 SPARK::TModifierCallback Class Reference

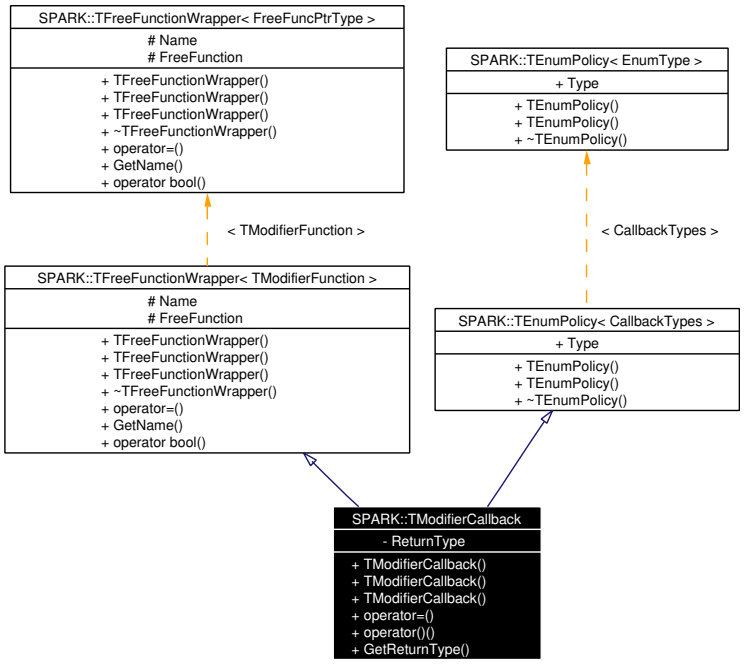
Function wrapper class for modifier callbacks.

`#include <callback.h>`

Inheritance diagram for SPARK::TModifierCallback:




Collaboration diagram for SPARK::TModifierCallback:



## Public Member Functions

- TModifierCallback (enum_policiy_type::enum_type code)

  *Constructor for a non-defined callback.*

- TModifierCallback (enum_policiy_type::enum_type code, const char *name, function_type ptr, ReturnTypes returnType)

  *Constructor for a defined callback.*

- TModifierCallback (const TModifierCallback &c)

  *Copy constructor.*

- TModifierCallback & operator= (const TModifierCallback &c)

  *Assignment operator.*

- void operator() (TObject *object, ArgList args, TargetList targets) const

  *Fires the underlying C function that implements the modifier callback.*

- ReturnTypes GetReturnType () const

  *Returns the return type for the modifier callback as ReturnTypes.*

### 3.12.1 Detailed Description

Function wrapper class for modifier callbacks.

### 3.12.2 Constructor & Destructor Documentation

#### 3.12.2.1 SPARK::TModifierCallback::TModifierCallback (enum_policiy_type::enum_type *code*) [inline]

Constructor for a non-defined callback.

#### 3.12.2.2 SPARK::TModifierCallback::TModifierCallback (enum_policiy_type::enum_type *code*, const char * *name*, function_type *ptr*, ReturnTypes *returnType*) [inline]

Constructor for a defined callback.

#### 3.12.2.3 SPARK::TModifierCallback::TModifierCallback (const TModifierCallback & *c*) [inline]

Copy constructor.

### 3.12.3 Member Function Documentation

#### 3.12.3.1 TModifierCallback& SPARK::TModifierCallback::operator= (const TModifierCallback & *c*) [inline]

Assignment operator.

#### 3.12.3.2 void SPARK::TModifierCallback::operator() (TObject * *object*, ArgList *args*, TargetList *targets*) const [inline]

Fires the underlying C function that implements the modifier callback.

**Parameters:**

***object*** is a pointer to the TObject instance the non-modifier callback belongs to

***args*** is the list of argument variable as defined in the FUNCTIONS {} statement of the atomic class

***targets*** is the list of target variable as defined in the FUNCTIONS {} statement of the atomic class

#### 3.12.3.3 ReturnTypes SPARK::TModifierCallback::GetReturnType () const [inline]

Returns the return type for the modifier callback as ReturnTypes.

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

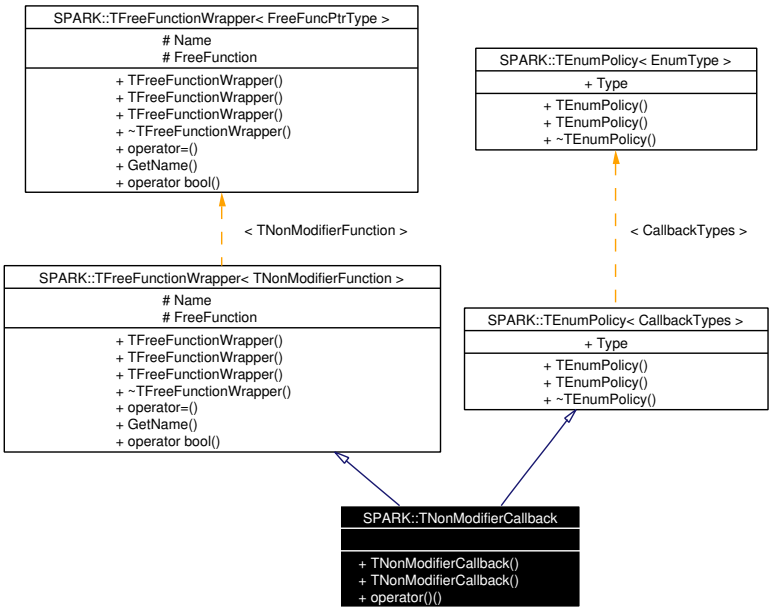
- callback.h

# 3.13 SPARK::TNonModifierCallback Class Reference

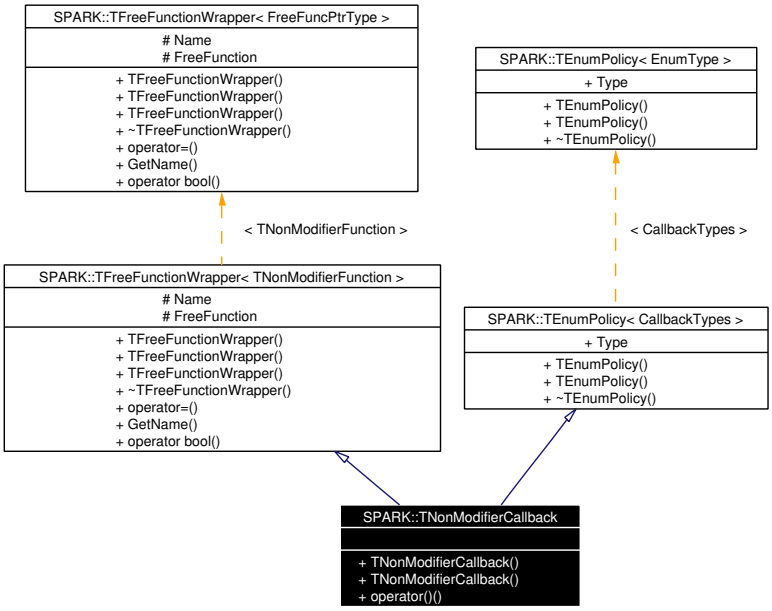
Function wrapper class for non-modifier callbacks.

`#include <callback.h>`

Inheritance diagram for SPARK::TNonModifierCallback:




Collaboration diagram for SPARK::TNonModifierCallback:



## Public Member Functions

- TNonModifierCallback (enum_policiy_type::enum_type code)
  *Constructor for a non-defined callback.*
- TNonModifierCallback (enum_policiy_type::enum_type code, const char *name, function_type ptr)
  *Constructor for a defined callback.*
- void operator() (TObject *object, ArgList args) const
  *Fires the underlying C function that implements the non-modifier callback.*

### 3.13.1 Detailed Description

Function wrapper class for non-modifier callbacks.

### 3.13.2 Constructor & Destructor Documentation

#### 3.13.2.1 SPARK::TNonModifierCallback::TNonModifierCallback (enum_policiy_type::enum_type *code*) [inline]

Constructor for a non-defined callback.

#### 3.13.2.2 SPARK::TNonModifierCallback::TNonModifierCallback (enum_policiy_type::enum_type *code*, const char * *name*, function_type *ptr*) [inline]

Constructor for a defined callback.

### 3.13.3 Member Function Documentation

#### 3.13.3.1 void SPARK::TNonModifierCallback::operator() (TObject * *object*, ArgList *args*) const [inline]

Fires the underlying C function that implements the non-modifier callback.

**Parameters:**

***object*** is a pointer to the TObject instance the non-modifier callback belongs to

***args*** is the list of arguments as defined in the FUNCTIONS {} statement of the atomic class

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

- callback.h

## 3.14 SPARK::TObject Class Reference

Class used to represent an instance of an inverse.

`#include <object.h>`

### Public Member Functions

#### Access methods

- unsigned GetHandle () const
  *Returns unique handle as specified in XML file.*
- unsigned GetInstanceHandle () const
  *Returns unique inverse handle set by SPARK::TInverse object for each instance.*
- const char * GetName () const
  *Returns name of object as specified in XML file as* `const char* .`
- const char * GetActiveCallbackName () const
  *Returns the name of the currently active callback, Returns "" if none active!*
- SPARK::TInverse * GetInverse ()
  *Returns name of SPARK::TInverse this object is an instance of.*
- const SPARK::TProblem * GetProblem () const
  *Returns pointer to SPARK::TProblem object this object belongs to.*
- const SPARK::TComponent * GetComponent () const
  *Returns pointer to SPARK::TComponent object this object belongs to.*
- double GetTolerance () const
  *Returns value of relative tolerance curently used by solver.*

#### Data methods

- void * GetData ()
  *Returns private data associated with this instance as* `void* .`
- void SetData (void *address)
  *Sets the private data to* `address`.

### 3.14.1 Detailed Description

Class used to represent an instance of an inverse.

**Note:**
Static callbacks do not require argument lists

### 3.14.2 Member Function Documentation

#### 3.14.2.1 unsigned SPARK::TObject::GetHandle () const [inline]

Returns unique handle as specified in XML file.

#### 3.14.2.2 unsigned SPARK::TObject::GetInstanceHandle () const [inline]

Returns unique inverse handle set by SPARK::TInverse object for each instance.

#### 3.14.2.3 const char* SPARK::TObject::GetName () const

Returns name of object as specified in XML file as `const char*` .

#### 3.14.2.4 const char* SPARK::TObject::GetActiveCallbackName () const

Returns the name of the currently active callback, Returns "" if none active!

#### 3.14.2.5 SPARK::TInverse* SPARK::TObject::GetInverse ()

Returns name of SPARK::TInverse this object is an instance of.

#### 3.14.2.6 const SPARK::TProblem* SPARK::TObject::GetProblem () const

Returns pointer to SPARK::TProblem object this object belongs to.

#### 3.14.2.7 const SPARK::TComponent* SPARK::TObject::GetComponent () const

Returns pointer to SPARK::TComponent object this object belongs to.

#### 3.14.2.8 double SPARK::TObject::GetTolerance () const

Returns value of relative tolerance curently used by solver.

#### 3.14.2.9 void* SPARK::TObject::GetData () [inline]

Returns private data associated with this instance as `void*` .

Here is the call graph for this function:


#### 3.14.2.10 void SPARK::TObject::SetData (void * *address*) [inline]

Sets the private data to `address`.

Here is the call graph for this function:


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

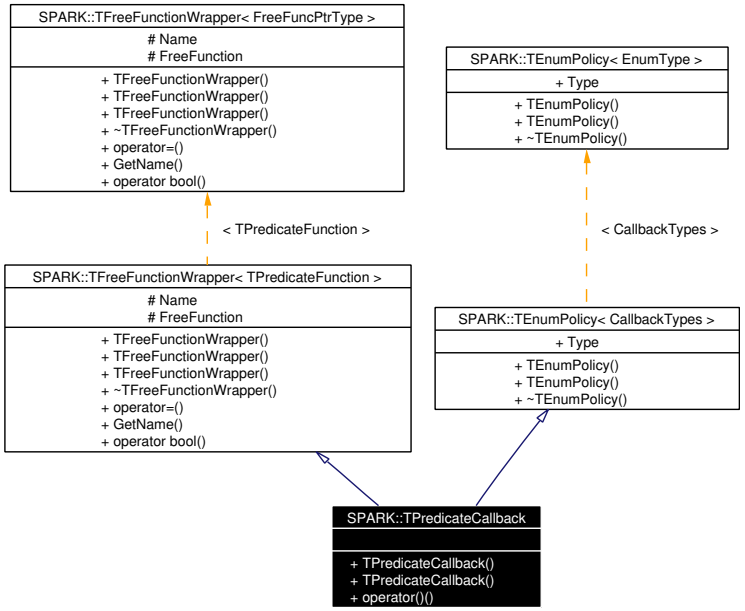
- object.h

# 3.15 SPARK::TPredicateCallback Class Reference

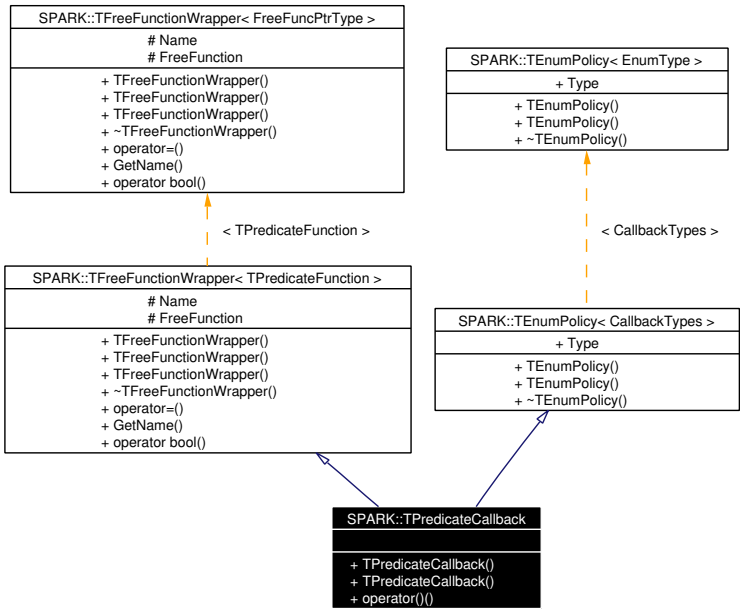
Function wrapper class for predicate callbacks.

`#include <callback.h>`

Inheritance diagram for SPARK::TPredicateCallback:




Collaboration diagram for SPARK::TPredicateCallback:



## Public Member Functions

- TPredicateCallback (enum_policiy_type::enum_type code)
  *Constructor for a non-defined callback.*
- TPredicateCallback (enum_policiy_type::enum_type code, const char *name, function_type ptr)
  *Constructor for a defined callback.*
- bool operator() (TObject *object, ArgList args) const
  *Fires the underlying C function that implements the predicate callback.*

### 3.15.1 Detailed Description

Function wrapper class for predicate callbacks.

### 3.15.2 Constructor & Destructor Documentation

#### 3.15.2.1 SPARK::TPredicateCallback::TPredicateCallback (enum_policiy_type::enum_type *code*) [inline]

Constructor for a non-defined callback.

#### 3.15.2.2 SPARK::TPredicateCallback::TPredicateCallback (enum_policiy_type::enum_type *code*, const char * *name*, function_type *ptr*) [inline]

Constructor for a defined callback.

### 3.15.3 Member Function Documentation

#### 3.15.3.1 bool SPARK::TPredicateCallback::operator() (TObject * *object*, ArgList *args*) const [inline]

Fires the underlying C function that implements the predicate callback.

**Parameters:**

***object*** is a pointer to the TObject instance the non-modifier callback belongs to

***args*** is the list of arguments as defined in the `FUNCTIONS {}` statement of the atomic class

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

- callback.h

## 3.16 SPARK::TProblem Class Reference

Representation of a problem object in the SPARK solver.

`#include <problem.h>`

### Public Types

- typedef std::bitset< DIAGNOSTICTYPES_L > TDiagnosticLevel

  *Bitset used to specify the diagnostic level with the different diagnostic types.*

- enum DiagnosticTypes {
  DiagnosticType_CONVERGENCE,
  DiagnosticType_INPUTS,
  DiagnosticType_REPORTS,
  DiagnosticType_PREFERENCES,
  DiagnosticType_STATISTICS,
  DiagnosticType_REQUESTS,
  DIAGNOSTICTYPES_L }

  *Level of more or less detailed diagnostic to the run log output stream.*

- enum SimulationFlags {
  SimulationFlag_OK = 0,
  SimulationFlag_BAD_NUMERICS,
  SimulationFlag_NO_CONVERGENCE,
  SimulationFlag_SINGULAR_SYSTEM,
  SimulationFlag_TIMESTEP_TOO_SMALL,
  SimulationFlag_FAILED_STEP,
  SimulationFlag_IDLE,
  SIMULATIONFLAGS_L }

  *Simulation flags.*

### Public Member Functions

#### Structors

- TProblem (unsigned numInverses, SPARK::TInverse *inverses[ ], unsigned numVariables, SPARK::TVariable *variables[ ], unsigned numComponents, SPARK::TComponent *components[ ]) throw (SPARK::XMemory)

  *Constructs a TProblem object from the structure specified by the variable arrays and the solution sequence described by the component array.*

- ~TProblem () throw ()

  *Destructor that frees memory allocated for the various solvers.*

#### Main methods invoked by driver function

- void Initialize (const SPARK::TRuntimeControls &controls) throw (SPARK::XMemory)

*Performs run-time initialization.*

- void LoadPreferenceSettings (const SPARK::TPreferenceSettings &preferences) throw (SPARK::XMemory)
  *Loads preference settings (default and for each component).*
- SimulationFlags Simulate () throw (SPARK::XInitialization)
  *Computes solution from InitialTime to FinalTime.*
- void Terminate ()
  *Ends processing and writes statistics to run log file.*

### State management functions

*These method let you save and restore the problem state in order to allow for a simulation restart from the saved state.*

- void Save (SPARK::TProblem::TState &state) const throw (SPARK::XInitialization)
  *Saves problem state at current time.*
- void Restore (const SPARK::TProblem::TState &state)
  *Restores problem state at the time specified in the state structure.*

### Access functions

- const char * GetName () const
  *Returns the name of the problem as const char*.*
- void SetName (const char *name)
  *Sets the problem name from name.*
- unsigned long GetStepCount () const
  *Returns the number of simulation steps performed so far.*
- SPARK::TGlobalSettings * GetGlobalSettings ()
  *Returns pointer to global settings object.*
- const SPARK::TGlobalSettings * GetGlobalSettings () const
  *Returns pointer to const global settings object.*

### Access operations for global problem variables

- const SPARK::TVariable & GetGlobalTime () const
  *Returns a const reference to the TVariable object that describes the global time link.*
- const SPARK::TVariable & GetGlobalTimeStep () const
  *Returns a const reference to the TVariable object that describes the global time step link.*

### Access operations for the problem variables

- SPARK::TVariable & GetVariable (unsigned handle) throw (SPARK::XAssertion)
- const SPARK::TVariable & GetVariable (unsigned handle) const throw (SPARK::XAssertion)
- SPARK::TVariable & GetVariable (const char *name) throw (SPARK::XAssertion)
- const SPARK::TVariable & GetVariable (const char *name) const throw (SPARK::XAssertion)

### Access operations for the problem inverses

- SPARK::TInverse * GetInverse (unsigned handle)
  *Returns pointer to TInverse object by handle.*
- SPARK::TInverse * GetInverse (const char *name)
  *Returns pointer to TInverse object by name.*

### Access operations for the problem objects

- SPARK::TObject * GetObject (unsigned handle)
  *Returns pointer to TObject object by handle.*
- SPARK::TObject * GetObject (const char *name)
  *Returns pointer to TObject object by name.*

### Predicate methods

- bool IsInitialTime () const
  *Returns true if global time is equal to initial time.*
- bool IsFinalTime () const
  *Returns true if global time is equal to final time.*
- bool Starting () const
- bool IsStaticStep () const
  *Returns true if current step is a static step.*
- bool IsTimeStepVariable () const
  *Returns true if key VariableTimeStep is set to 1 in runtime controls.*
- bool IsReady () const
  *Returns true if problem is ready for simulation. False otherwise.*
- bool IsDiagnostic () const
  *Returns true if any diagnostic is set.*
- bool IsDiagnostic (DiagnosticTypes d) const
  *Returns true if diagnostic d is set.*

### IO functions

- bool WriteStamp (std::ostream &os) const
  *Writes current step stamp to output stream os.*
- void ReportStatistics (std::ostream &os, const std::string &before) const
  *Writes simulation statistics to output stream os.*
- void GenerateSnapshot (const std::string &filename) const throw (SPARK::XInitialization)
  *Generates the snapshot file named "filename".*

### 3.16.1 Detailed Description

Representation of a problem object in the SPARK solver.

Class methods implement :

- read values from input files
- generate snapshot files and write reported values to output or trace files
- solve system of equations for the unknown problem variables at $Clock = t_{initial}$ where $t_{initial} = InitialTime$ as specified in the runtime controls
- solve system of equations for the unknown problem variables for $t_{initial} + dt <= t <= t_{final}$ where $dt = TimeStep$ and $t_{final} = FinalTime$ as specified in the runtime controls file
- save current values of problem variables as history

**Note:**

The TProblem class relies for all runtime information on a SPARK::TRuntimeControls object that is passed to the TProblem::Initialize() method.

- Invoke the TProblem::Initialize() method to bind the SPARK::TRuntimeControls object with a TProblem object.
- Before re-invoking the TProblem::Initialize() method, make sure that you have invoked the method TProblem::Terminate() first to reset the various managers.

### 3.16.2 Member Typedef Documentation

#### 3.16.2.1 typedef std::bitset<DIAGNOSTICTYPES_L> SPARK::TProblem::TDiagnosticLevel

Bitset used to specify the diagnostic level with the different diagnostic types.

To make a run with convergence and preferences diagnostic, specifiy `DiagnosticLevel( 9 () )` in the runtime controls file whereby 9 = 1 (*DiagnosticType_CONVERGENCE*) + 8 (*DiagnosticType_PREFERENCES*)

### 3.16.3 Member Enumeration Documentation

#### 3.16.3.1 enum SPARK::TProblem::DiagnosticTypes

Level of more or less detailed diagnostic to the run log output stream.

**Enumeration values:**

***DiagnosticType_CONVERGENCE*** (=1) show convergence behavior at each iteration

***DiagnosticType_INPUTS*** (=2) show where the variables get their input from

***DiagnosticType_REPORTS*** (=4) show reported values at each report event

***DiagnosticType_PREFERENCES*** (=8) show preference settings as loaded in LoadPreferenceSettings()

***DiagnosticType_STATISTICS*** (=16) show run statistics at the end of simulation

***DiagnosticType_REQUESTS*** (=32) show atomic class requests

***DIAGNOSTICTYPES_L*** Number of diagnostic types.

#### 3.16.3.2 enum SPARK::TProblem::SimulationFlags

Simulation flags.

**Enumeration values:**

***SimulationFlag_OK*** Step was computed successfully.

***SimulationFlag_BAD_NUMERICS*** Bad numerics detected.

***SimulationFlag_NO_CONVERGENCE*** Could not obtain convergence while solving nonlinear system(s).

***SimulationFlag_SINGULAR_SYSTEM*** Detected a singular or "badly-conditioned" linear system.

***SimulationFlag_TIMESTEP_TOO_SMALL*** Time step selection is limited by MinTimeStep from runtime controls file.

***SimulationFlag_FAILED_STEP*** Step was rejected too many times in response to user's requests.

***SimulationFlag_IDLE*** Default step when starting the simulation.

***SIMULATIONFLAGS_L*** Number of simulation flags.

## 3.16.4 Constructor & Destructor Documentation

#### 3.16.4.1 SPARK::TProblem::TProblem (unsigned *numInverses*, SPARK::TInverse * *inverses*[ ], unsigned *numVariables*, SPARK::TVariable * *variables*[ ], unsigned *numComponents*, SPARK::TComponent * *components*[ ]) throw (SPARK::XMemory)

Constructs a TProblem object from the structure specified by the variable arrays and the solution sequence described by the component array.

**Parameters:**

***numInverses*** Number of inverses

***inverses*** Array of pointers to TInverse objects

***numVariables*** Number of problem variables

***variables*** Array of pointers to TVariable objects

***numComponents*** Number of components comprising the problem

***components*** Array of pointers to TComponent objects

**Exceptions:**

***Throws*** SPARK::XMemory if out of memory when building the problem.

#### 3.16.4.2 SPARK::TProblem::~TProblem () throw ()

Destructor that frees memory allocated for the various solvers.

**Note:**

Destructor does not destroy any data that was passed to the constructor to describe the problem topology. The calling program is responsible for destroying these data structures explicitly if needed.

## 3.16.5 Member Function Documentation

#### 3.16.5.1 void SPARK::TProblem::Initialize (const SPARK::TRuntimeControls & *controls*) throw (SPARK::XMemory)

Performs run-time initialization.

**Parameters:**
***controls*** runtime control parameters for the simulation

**Postcondition:**
Initializes clock manager
Allocates history and inner values
Prepares input and output managers
Loads initial values
Fires construct callbacks

#### 3.16.5.2 void SPARK::TProblem::LoadPreferenceSettings (const SPARK::TPreferenceSettings & *preferences*) throw (SPARK::XMemory)

Loads preference settings (default and for each component).

**Parameters:**
***preferences*** An object of class SPARK::TPreferenceSettings that contains data from the ∗.prf file

**Postcondition:**
The SPARK::TGlobalSettings object is loaded for this problem.
The SPARK::TComponentSettings object are loaded for each component.

#### 3.16.5.3 SimulationFlags SPARK::TProblem::Simulate () throw (SPARK::XInitialization)

Computes solution from InitialTime to FinalTime.

**Returns:**
Simulation flag as enum SimulationFlags

**Precondition:**
Problem must have initialized with a prior call to TProblem::Initialize()

**Postcondition:**
Output and trace files are updated after each simulation step until end of simulation.
Initial snapshot file is generated after the first step if requested in the runtime controls file.
Final snapshot file is generated at the last step if requested in the runtime controls file.

**Exceptions:**
***Throws*** a SPARK::XInitialization exception object if problem was not initialized or if it is in an inconsistent state.

#### 3.16.5.4 void SPARK::TProblem::Terminate ()

Ends processing and writes statistics to run log file.

**Postcondition:**
Fires destruct callbacks
Deletes input and output managers
Resets internal counters to allow for a fresh simulation run with a new set of runtime controls specified with a call to TProblem::Initialize().

#### 3.16.5.5 void SPARK::TProblem::Save (SPARK::TProblem::TState & *state*) const throw (SPARK::XInitialization)

Saves problem state at current time.

This method populates the state object with the state of the problem at the current time.

**Parameters:**

***state*** Object containing the problem state being saved.

**Exceptions:**

***Throws*** a SPARK::XInitialization exception object if problem could not be saved

#### 3.16.5.6 void SPARK::TProblem::Restore (const SPARK::TProblem::TState & *state*)

Restores problem state at the time specified in the state structure.

The method initializes the variables with the trajectory values stored in the state object. For now there is no support for class data persistency.

**Parameters:**

***state*** Object containing the saved problem state to be restored.

#### 3.16.5.7 const char* SPARK::TProblem::GetName () const

Returns the name of the problem as const char*.

#### 3.16.5.8 void SPARK::TProblem::SetName (const char * *name*)

Sets the problem name from name.

#### 3.16.5.9 unsigned long SPARK::TProblem::GetStepCount () const

Returns the number of simulation steps performed so far.

#### 3.16.5.10 SPARK::TGlobalSettings* SPARK::TProblem::GetGlobalSettings ()

Returns pointer to global settings object.

#### 3.16.5.11 const SPARK::TGlobalSettings* SPARK::TProblem::GetGlobalSettings () const

Returns pointer to const global settings object.

#### 3.16.5.12 const SPARK::TVariable& SPARK::TProblem::GetGlobalTime () const

Returns a const reference to the TVariable object that describes the global time link.

#### 3.16.5.13 const SPARK::TVariable& SPARK::TProblem::GetGlobalTimeStep () const

Returns a const reference to the TVariable object that describes the global time step link.

#### 3.16.5.14 SPARK::TVariable& SPARK::TProblem::GetVariable (unsigned *handle*) throw (SPARK::XAssertion)

Returns TVariable& object by handle

**Parameters:**
***handle*** is the unsigned value that uniquely identifies this variable as specified in the XML file. See TVariable::Handle

**Exceptions:**
***SPARK::XAssertion*** Could not find a variable for this handle

#### 3.16.5.15 const SPARK::TVariable& SPARK::TProblem::GetVariable (unsigned *handle*) const throw (SPARK::XAssertion)

Returns const TVariable& object by handle

**Parameters:**
***handle*** is the unsigned value that uniquely identifies this variable as specified in the XML file. See TVariable::Handle

**Exceptions:**
***SPARK::XAssertion*** Could not find a variable for this handle

#### 3.16.5.16 SPARK::TVariable& SPARK::TProblem::GetVariable (const char * *name*) throw (SPARK::XAssertion)

Returns TVariable& object by name

**Parameters:**
***name*** const char* that stands for the name of the variable as specified in the XML file. See TVariable::Name

**Note:**
The variable name is case-sensitive.

**Exceptions:**
***SPARK::XAssertion*** Could not find a variable with this name.

#### 3.16.5.17 const SPARK::TVariable& SPARK::TProblem::GetVariable (const char * *name*) const throw (SPARK::XAssertion)

Returns const TVariable& object by name

**Parameters:**
***name*** const char* that stands for the name of the variable as specified in the XML file. See TVariable::Name

**Note:**
The variable name is case-sensitive.

**Exceptions:**
***SPARK::XAssertion*** Could not find a variable with this name.

#### 3.16.5.18 SPARK::TInverse* SPARK::TProblem::GetInverse (unsigned *handle*)

Returns pointer to TInverse object by handle.

#### 3.16.5.19 SPARK::TInverse* SPARK::TProblem::GetInverse (const char * *name*)

Returns pointer to TInverse object by name.

#### 3.16.5.20 SPARK::TObject* SPARK::TProblem::GetObject (unsigned *handle*)

Returns pointer to TObject object by handle.

#### 3.16.5.21 SPARK::TObject* SPARK::TProblem::GetObject (const char * *name*)

Returns pointer to TObject object by name.

#### 3.16.5.22 bool SPARK::TProblem::IsInitialTime () const

Returns true if global time is equal to initial time.

#### 3.16.5.23 bool SPARK::TProblem::IsFinalTime () const

Returns true if global time is equal to final time.

#### 3.16.5.24 bool SPARK::TProblem::Starting () const

Returns true if first step of simulation.

**Note:**

True for first step computed when calling TProblem::Simulate() This is different than checking for GetStepCount()==1 since the StepCount counter is incremented at every step, possibly for multiple calls to TProblem::Simulate(). StepCount is reset to 0 only when calling TProblem::Initialize().

#### 3.16.5.25 bool SPARK::TProblem::IsStaticStep () const

Returns true if current step is a static step.

#### 3.16.5.26 bool SPARK::TProblem::IsTimeStepVariable () const

Returns true if key VariableTimeStep is set to 1 in runtime controls.

#### 3.16.5.27 bool SPARK::TProblem::IsReady () const

Returns true if problem is ready for simulation. False otherwise.

#### 3.16.5.28 bool SPARK::TProblem::IsDiagnostic () const [inline]

Returns true if any diagnostic is set.

#### 3.16.5.29 bool SPARK::TProblem::IsDiagnostic (DiagnosticTypes *d*) const [inline]

Returns true if diagnostic d is set.

#### 3.16.5.30 bool SPARK::TProblem::WriteStamp (std::ostream & *os*) const

Writes current step stamp to output stream os.

#### 3.16.5.31 void SPARK::TProblem::ReportStatistics (std::ostream & *os*, const std::string & *before*) const

Writes simulation statistics to output stream os.

#### 3.16.5.32 void SPARK::TProblem::GenerateSnapshot (const std::string & *filename*) const throw (SPARK::XInitialization)

Generates the snapshot file named "filename".

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

- problem.h

## 3.17 SPARK::TProblem::TState Class Reference

Interface class defining the methods used to save and restore the state of the problem using the TProblem::Save() and TProblem::Restore() methods.

`#include <problem.h>`

### Public Member Functions

#### Structors

- TState ()
  *Default consrtuctor.*
- TState (const TState &)
  *Copy constructor: deep copy of values and class states.*
- ∼TState () throw ()
  *Destructor.*

#### Access functions for trajectory

- const char ∗ GetContext () const
  *Returns the character string that dscribes the context at the time the problem state was saved.*
- double GetTime (unsigned idx) const
  *Returns the value of the global time variable for the past value* `idx`.
- double GetTimeStep (unsigned idx) const
  *Returns the value of the global time step variable for the past value* `idx`.
- unsigned GetNumVariables () const
  *Returns the number of variables stored in Trajectory.*
- unsigned GetNumPastValues () const
  *Returns the number of past values stored in Trajectory for each variable.*
- value_container & GetPastValues (unsigned idx)
  *Returns container with desired past values corresponding to index* `idx`.
- const value_container & GetPastValues (unsigned idx) const
  *Returns container with desired past values corresponding to index* `idx`.

#### Main operations

- void Save (const std::string &context, const SPARK::TTrajectory &trajectory, const TTopology &topology)
  *Loads the problem state at the current time (values and class data).*
- void Restore (TTrajectory &trajectory, TTopology &topology) const
  *Restores the problem state.*

### 3.17.1 Detailed Description

Interface class defining the methods used to save and restore the state of the problem using the TProblem::Save() and TProblem::Restore() methods.

The class defines methods used to save the values of all variables at the current global time. Then an instance of this class can be used to restore the problem state by invoking TProblem::Restore() in order to restart a simulation.

**Warning:**

These methods do not try to store or load the private data associated with each inverse and/or object in the problem under study. This can potentially cause problems when restarting the simulation with saved solution values because the private data might be in a different state. A later version of SPARK will provide support for full serialization of the problem state, i.e., including class private data, through the addition of dedicated callbacks.

### 3.17.2 Constructor & Destructor Documentation

#### 3.17.2.1 SPARK::TProblem::TState::TState ()

Default consrtuctor.

#### 3.17.2.2 SPARK::TProblem::TState::TState (const TState &)

Copy constructor: deep copy of values and class states.

#### 3.17.2.3 SPARK::TProblem::TState::~TState () throw ()

Destructor.

### 3.17.3 Member Function Documentation

#### 3.17.3.1 const char* SPARK::TProblem::TState::GetContext () const

Returns the character string that dscribes the context at the time the problem state was saved.

#### 3.17.3.2 double SPARK::TProblem::TState::GetTime (unsigned *idx*) const

Returns the value of the global time variable for the past value `idx`.

**Exceptions:**

***Throws*** SPARK::Xinitialization if operation failed

#### 3.17.3.3 double SPARK::TProblem::TState::GetTimeStep (unsigned *idx*) const

Returns the value of the global time step variable for the past value `idx`.

**Exceptions:**

***Throws*** SPARK::Xinitialization if operation failed

#### 3.17.3.4 unsigned SPARK::TProblem::TState::GetNumVariables () const

Returns the number of variables stored in Trajectory.

**Returns:**
Number of values defining the trajectory, including the values of the global time and the global time step.

#### 3.17.3.5 unsigned SPARK::TProblem::TState::GetNumPastValues () const

Returns the number of past values stored in Trajectory for each variable.

**Returns:**
Number of past values

#### 3.17.3.6 value_container& SPARK::TProblem::TState::GetPastValues (unsigned *idx*)

Returns container with desired past values corresponding to index `idx`.

**Returns:**
Reference to container of values

#### 3.17.3.7 const value_container& SPARK::TProblem::TState::GetPastValues (unsigned *idx*) const

Returns container with desired past values corresponding to index `idx`.

**Returns:**
Const reference to container of values

#### 3.17.3.8 void SPARK::TProblem::TState::Save (const std::string & *context*, const SPARK::TTrajectory & *trajectory*, const TTopology & *topology*)

Loads the problem state at the current time (values and class data).

If necessary, frees previously allocated memory.

**Parameters:**
***context*** Description of the problem context
***trajectory*** Trajectory tracker for the current problem
***topology*** Problem topology to save

#### 3.17.3.9 void SPARK::TProblem::TState::Restore (TTrajectory & *trajectory*, TTopology & *topology*) const

Restores the problem state.

**Parameters:**
***trajectory*** Trajectory tracker for the current problem
***topology*** Problem topology to restore

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

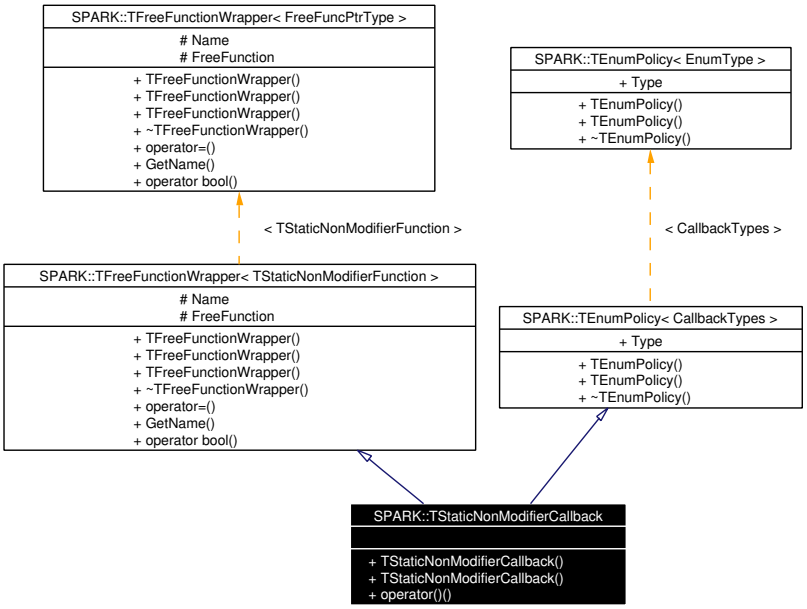
- problem.h

## 3.18 SPARK::TStaticNonModifierCallback Class Reference

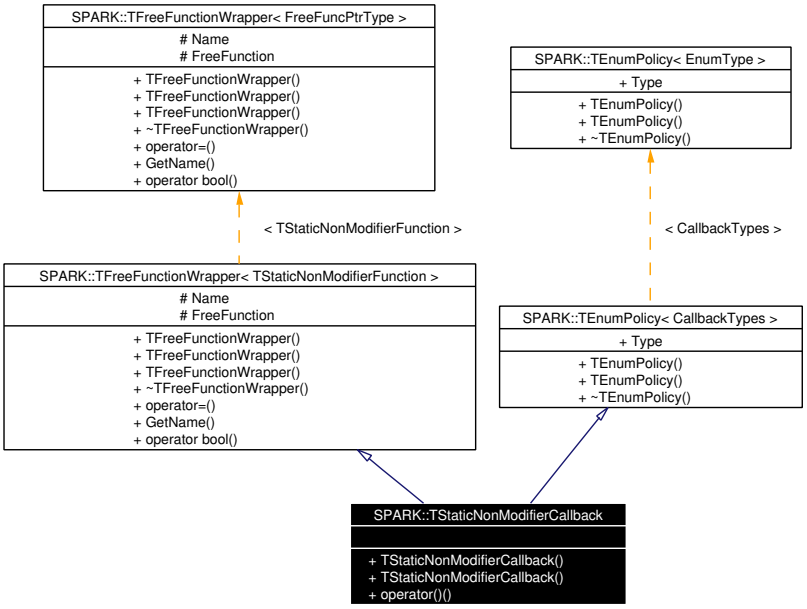
Function wrapper class for static non-modifier callbacks.

`#include <callback.h>`

Inheritance diagram for SPARK::TStaticNonModifierCallback:




Collaboration diagram for SPARK::TStaticNonModifierCallback:



## Public Member Functions

- TStaticNonModifierCallback (enum_policiy_type::enum_type code)
  *Constructor for a non-defined callback.*
- TStaticNonModifierCallback (enum_policiy_type::enum_type code, const char *name, function_type ptr)
  *Constructor for a defined callback.*
- void operator() (TInverse *inverse) const
  *Fires the underlying C function that implements the static callback.*

### 3.18.1 Detailed Description

Function wrapper class for static non-modifier callbacks.

### 3.18.2 Constructor & Destructor Documentation

#### 3.18.2.1 SPARK::TStaticNonModifierCallback::TStaticNonModifierCallback (enum_policiy_type::enum_type *code*) [inline]

Constructor for a non-defined callback.

#### 3.18.2.2 SPARK::TStaticNonModifierCallback::TStaticNonModifierCallback (enum_policiy_type::enum_type *code*, const char * *name*, function_type *ptr*) [inline]

Constructor for a defined callback.

### 3.18.3 Member Function Documentation

#### 3.18.3.1 void SPARK::TStaticNonModifierCallback::operator() (TInverse * *inverse*) const [inline]

Fires the underlying C function that implements the static callback.

**Parameters:**

***inverse*** is a pointer to the TInverse instance the static callback belongs to

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

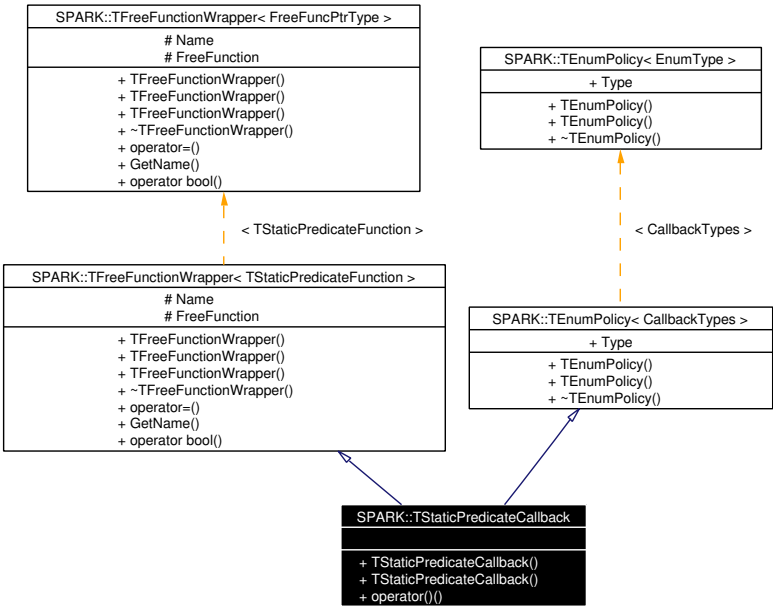
- callback.h

## 3.19 SPARK::TStaticPredicateCallback Class Reference

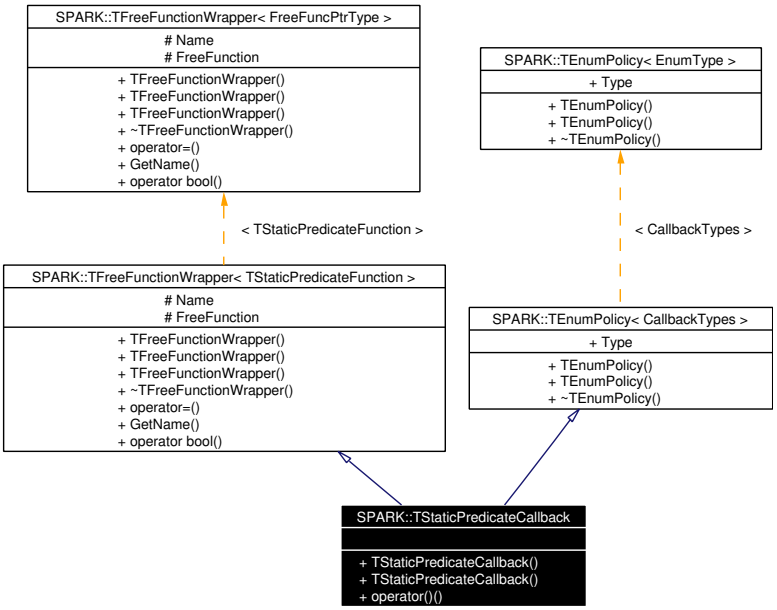
Function wrapper class for static predicate callbacks.

`#include <callback.h>`

Inheritance diagram for SPARK::TStaticPredicateCallback:




Collaboration diagram for SPARK::TStaticPredicateCallback:



## Public Member Functions

- TStaticPredicateCallback (enum_policiy_type::enum_type code)
  *Constructor for a non-defined callback.*
- TStaticPredicateCallback (enum_policiy_type::enum_type code, const char *name, function_type ptr)
  *Constructor for a defined callback.*
- bool operator() (TInverse *inverse) const
  *Fires the underlying C function that implements the static predicate callback.*

### 3.19.1 Detailed Description

Function wrapper class for static predicate callbacks.

### 3.19.2 Constructor & Destructor Documentation

#### 3.19.2.1 SPARK::TStaticPredicateCallback::TStaticPredicateCallback (enum_policiy_type::enum_type *code*) [inline]

Constructor for a non-defined callback.

#### 3.19.2.2 SPARK::TStaticPredicateCallback::TStaticPredicateCallback (enum_policiy_type::enum_type *code*, const char * *name*, function_type *ptr*) [inline]

Constructor for a defined callback.

### 3.19.3 Member Function Documentation

#### 3.19.3.1 bool SPARK::TStaticPredicateCallback::operator() (TInverse * *inverse*) const [inline]

Fires the underlying C function that implements the static predicate callback.

**Parameters:**

***inverse*** is a pointer to the TInverse instance the static predicate callback belongs to

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

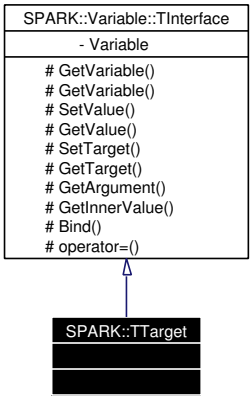
- callback.h

## 3.20 SPARK::TTarget Class Reference

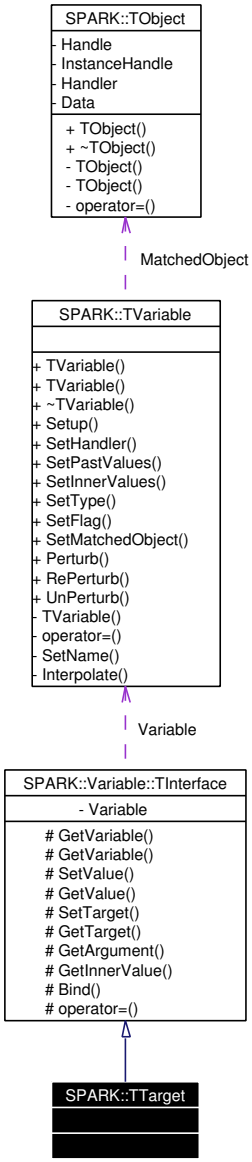
This class acts as a write-only interface to a TVariable object. It is used only in the callbacks to describe the target variables. This class is not used internally in the solver. It provides proper value access behavior depending on the type of the variable and the calling context. Also, it implements some of the attribute access methods from TVariable.

`#include <callback.h>`

Inheritance diagram for SPARK::TTarget:




Collaboration diagram for SPARK::TTarget:




## Public Member Functions

### Structors

- TTarget ()
  *Default constructor.*
- TTarget (SPARK::TVariable *variable)
  *Normal constructor using a pointer to a TVariable object.*
- TTarget (const TTarget &target)
  *Copy constructor.*

### Numerical access methods

*The overloaded arithmetic operators operate on double only and are all unary operators.*

- TTarget & **operator=** (double scalar)
- TTarget & **operator+=** (double scalar)
- TTarget & **operator-=** (double scalar)
- TTarget & **operator** ∗**=** (double scalar)
- TTarget & **operator/=** (double scalar)

### 3.20.1 Detailed Description

This class acts as a write-only interface to a TVariable object. It is used only in the callbacks to describe the target variables. This class is not used internally in the solver. It provides proper value access behavior depending on the type of the variable and the calling context. Also, it implements some of the attribute access methods from TVariable.

### 3.20.2 Constructor & Destructor Documentation

#### 3.20.2.1 SPARK::TTarget::TTarget () [inline]

Default constructor.

#### 3.20.2.2 SPARK::TTarget::TTarget (SPARK::TVariable ∗ *variable*) [inline, explicit]

Normal constructor using a pointer to a TVariable object.

#### 3.20.2.3 SPARK::TTarget::TTarget (const TTarget & *target*) [inline]

Copy constructor.

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

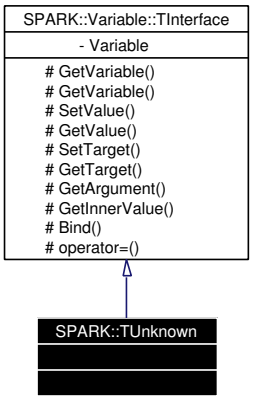
- callback.h

## 3.21 SPARK::TUnknown Class Reference

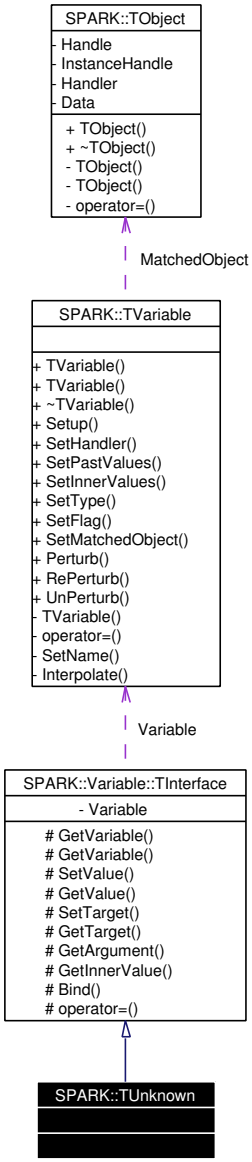
Class used to represent the properties and the numerical values of an unknown problem variable.

`#include <variable.h>`

Inheritance diagram for SPARK::TUnknown:




Collaboration diagram for SPARK::TUnknown:




## Public Member Functions

### Structors

- TUnknown ()
  *Default constructor.*
- TUnknown (SPARK::TVariable *variable)
  *Explicit constructor.*
- TUnknown (const TUnknown &unknown)
  *Copy constructor.*
- TUnknown & operator= (const TUnknown &u)

### Main operations (used to switch between calling contexts internally)

- void Perturb ()
  *Prepares the interface for a perturbation.*
- void RePerturb ()
  *Prepares the interface for a re-perturbation.*
- void UnPerturb ()
  *Unperturb the interface after various perturbations.*

**Access methods**

- SPARK::TVariable & operator ∗ ()
  *Dereferencing operator.*
- const SPARK::TVariable & operator ∗ () const
  *Const flavor of dereferencing operator.*
- SPARK::TVariable ∗ operator → ()
  *Indirection operator.*
- const SPARK::TVariable ∗ operator → () const
  *Const flavor of indirection operator.*
- double GetTarget () const
  *Returns value from evaluating the matched object (used by TEquationSystem and TJacobian internally).*
- void SetScale (double scalar)
  *Sets the current scale to scalar.*

**Numerical access and arithmetic unary operators**

*The overloaded arithmetic operators operate on double only and are all unary operators.*

- **operator double** () const
- TUnknown & **operator=** (double scalar)
- TUnknown & **operator+=** (double scalar)
- TUnknown & **operator-=** (double scalar)
- TUnknown & **operator** ∗**=** (double scalar)
- TUnknown & **operator/=** (double scalar)

**Predicate methods**

## 3.21.1 Detailed Description

Class used to represent the properties and the numerical values of an unknown problem variable.

Interface for internal manipulation of a SPARK::TVariable instance when it is an unknown. Dereference underlying SPARK::TVariable∗ and provides numeric interface for basic arithmetic.

## 3.21.2 Constructor & Destructor Documentation

### 3.21.2.1 SPARK::TUnknown::TUnknown ()

Default constructor.

#### 3.21.2.2 SPARK::TUnknown::TUnknown (SPARK::TVariable * *variable*) [explicit]

Explicit constructor.

#### 3.21.2.3 SPARK::TUnknown::TUnknown (const TUnknown & *unknown*)

Copy constructor.

### 3.21.3 Member Function Documentation

#### 3.21.3.1 TUnknown& SPARK::TUnknown::operator= (const TUnknown & *u*)

Assignment operator

**Warning:**
Use carefully. It works only if the interface has not been initialized yet.

**Parameters:**
***u*** Another TUnknown object used to initialize the underlying pointer of this TUnknown object

**Exceptions:**
***SPARK::XInitialization*** Thrown if this object has already been initialized before this call.

#### 3.21.3.2 void SPARK::TUnknown::Perturb ()

Prepares the interface for a perturbation.

#### 3.21.3.3 void SPARK::TUnknown::RePerturb ()

Prepares the interface for a re-perturbation.

#### 3.21.3.4 void SPARK::TUnknown::UnPerturb ()

Unperturb the interface after various perturbations.

#### 3.21.3.5 SPARK::TVariable& SPARK::TUnknown::operator * () [inline]

Dereferencing operator.

#### 3.21.3.6 const SPARK::TVariable& SPARK::TUnknown::operator * () const [inline]

Const flavor of dereferencing operator.

#### 3.21.3.7 SPARK::TVariable* SPARK::TUnknown::operator -> () [inline]

Indirection operator.

#### 3.21.3.8 const SPARK::TVariable* SPARK::TUnknown::operator → () const [inline]

Const flavor of indirection operator.

#### 3.21.3.9 double SPARK::TUnknown::GetTarget () const

Returns value from evaluating the matched object (used by TEquationSystem and TJacobian internally).

Reimplemented from SPARK::Variable::TInterface.

#### 3.21.3.10 void SPARK::TUnknown::SetScale (double *scalar*)

Sets the current scale to scalar.

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

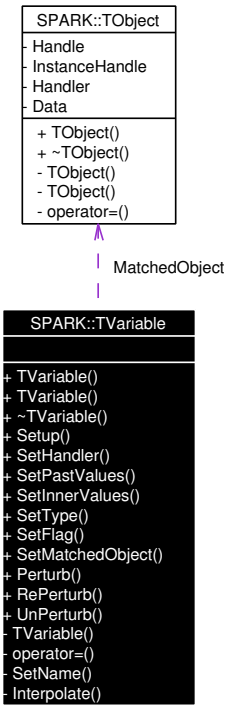
- variable.h

## 3.22 SPARK::TVariable Class Reference

Class used to represent the properties and the numerical values of a problem variable.

`#include <variable.h>`

Collaboration diagram for SPARK::TVariable:




### Public Types

- enum FlagTypes {
  FlagType_BREAK = 0,
  FlagType_RESIDUAL,
  FlagType_DYNAMIC,
  FlagType_INPUT_FROM_LINK,
  FlagType_PREDICT_FROM_LINK,
  FlagType_REPORT,
  FLAGTYPES_L }

  *Describes internal properties of the variable as specified by setupcpp.*

### Public Member Functions

- TVariable ()

  *Default constructor used when loading the problem at runtime.*

- TVariable (unsigned handle, const char ∗name, const char ∗unit, SPARK::VariableTypes type, double initVal, double minVal, double maxVal, double atol, unsigned fromLinkId, const char ∗rd_url, const char ∗wr_url)

*Constructor used with compiled "problem.cpp" file.*

### Access methods for the current numerical value of the variable

- double GetValue () const
  *Returns the curent value as a double.*
- operator double () const
  *Enables implicit conversion to (double).*
- void SetValue (double scalar)
  *Sets the current value to scalar.*
- SPARK::TVariable & operator= (double scalar)
  *Sets the current value to scalar with operator=(double ).*
- double GetArgument () const
  *Returns value for argument variable in an object.*
- void SetTarget (double scalar)
  *Sets the target value for the matched object.*
- double GetTarget () const
  *Returns value from evaluating the matched object (used by TEquationSystem and TJacobian internally).*
- double Predict () const
  *Predicts value for current time based on past values only.*

### Access methods for past and inner values

- unsigned GetNumPastValues () const
  *Returns the number of past values that are being kept track of.*
- double GetPastValue (unsigned pastStep) const
  *Returns the past value from idx steps ago.*
- double operator[ ] (unsigned pastStep) const
  *Returns the past value from idx steps ago.*
- double GetInnerValue (unsigned innerStep) const
  *Access the intermediate values over the currrent step.*
- double operator() (unsigned innerStep) const
  *Access the intermediate values over the currrent step.*

### Updating methods

- void SetFromLinkId (unsigned var_id)
  *Set INPUT_FROM_LINK=FromLink or PREDICT_FROM_LINK=FromLink.*
- unsigned GetFromLinkId () const
  *Get var_id of FromLink variables, SPARK::NotAssigned if not specified.*

### Access operations for the internal properties

- TObject ∗ GetMatchedObject ()
- const TObject ∗ GetMatchedObject () const

### Access operations for the LINK properties

- unsigned GetHandle () const
  *Returns the unique handle as unsigned int.*
- SPARK::VariableTypes GetType () const
  *Returns the variable type as SPARK::VariableTypes.*
- unsigned long GetFlag () const
  *Returns the variable flag as an unsigned int.*
- const std::string & GetName () const
  *Returns the variable name as a const char∗.*
- const std::string & GetUnit () const
  *Returns the unit string as const char∗.*
- void SetUnit (const char ∗unit)
  *Sets the unit string to unit.*
- double GetInit () const
  *Returns the initial value as double.*
- void SetInit (double init)
  *Sets the initial value to init.*
- double GetMin () const
  *Returns the mimimum value as double.*
- void SetMin (double min)
  *Sets the minimum value to min.*
- double GetMax () const
  *Returns the maximum value as double.*
- void SetMax (double max)
  *Sets the maximum value to max.*
- double GetAbsTolerance () const
  *Returns the absolute tolerance as double.*
- void SetAbsTolerance (double atol)
  *Sets the absolute tolerance to atol.*
- double GetScale () const
  *Returns the current scale.*
- void SetScale (double scalar)
  *Sets the current scale to scalar.*

### Predicate methods

- bool IsFlag (FlagTypes flag) const
  *Returns if flag is set for this variable, false otherwise.*

**IO operations**

- void Write (std::ostream &os, unsigned width, unsigned precision) const

**R/W URL methods**

***Note:***
*Not yet documented*

- const char * **GetReadUrlString** () const
- void **SetReadUrlString** (const char *url)
- int **GetReadUrlHandle** () const
- void **SetReadUrlHandle** (int s)
- const char * **GetWriteUrlString** () const
- void **SetWriteUrlString** (const char *url)
- int **GetWriteUrlHandle** () const
- void **SetWriteUrlHandle** (int s)

## 3.22.1 Detailed Description

Class used to represent the properties and the numerical values of a problem variable.

This class is the building block of the SPARK solver framework. It is used to represent all problem variables at the problem and component levels.

The low-level numerical solution methods do not operate directly on the SPARK::TVariable instances but on arrays of fundamental types (i.e., double, unsigned...).

Eventually, the numerical values in the SPARK::TVariable objects are updated with the solution of the nonlinear solver.

## 3.22.2 Member Enumeration Documentation

### 3.22.2.1 enum SPARK::TVariable::FlagTypes

Describes internal properties of the variable as specified by setupcpp.

**Enumeration values:**

***FlagType_BREAK*** setupcpp flaged this LINK as a break variable

***FlagType_RESIDUAL*** matched EVALUATE callback returns RESIDUAL value

***FlagType_DYNAMIC*** connected to X port of an INTEGRATOR atomic class

***FlagType_INPUT_FROM_LINK*** LINK specified with INPUT_FROM_LINK.

***FlagType_PREDICT_FROM_LINK*** LINK specified with PREDICT_FROM_LINK.

***FlagType_REPORT*** LINK specified with REPORT or Write URL specified with REPORT tag.

***FLAGTYPES_L*** Number of different flags.

## 3.22.3 Constructor & Destructor Documentation

### 3.22.3.1 SPARK::TVariable::TVariable ()

Default constructor used when loading the problem at runtime.

#### 3.22.3.2 SPARK::TVariable::TVariable (unsigned *handle*, const char * *name*, const char * *unit*, SPARK::VariableTypes *type*, double *initVal*, double *minVal*, double *maxVal*, double *atol*, unsigned *fromLinkId*, const char * *rd_url*, const char * *wr_url*)

Constructor used with compiled "problem.cpp" file.

## 3.22.4 Member Function Documentation

#### 3.22.4.1 double SPARK::TVariable::GetValue () const [inline]

Returns the curent value as a double.

#### 3.22.4.2 SPARK::TVariable::operator double () const [inline]

Enables implicit conversion to (double).

#### 3.22.4.3 void SPARK::TVariable::SetValue (double *scalar*) [inline]

Sets the current value to scalar.

#### 3.22.4.4 SPARK::TVariable& SPARK::TVariable::operator= (double *scalar*) [inline]

Sets the current value to scalar with operator=(double ).

#### 3.22.4.5 double SPARK::TVariable::GetArgument () const [inline]

Returns value for argument variable in an object.

#### 3.22.4.6 void SPARK::TVariable::SetTarget (double *scalar*) [inline]

Sets the target value for the matched object.

#### 3.22.4.7 double SPARK::TVariable::GetTarget () const [inline]

Returns value from evaluating the matched object (used by TEquationSystem and TJacobian internally).

#### 3.22.4.8 double SPARK::TVariable::Predict () const

Predicts value for current time based on past values only.

Returns an estimate of the solution at the current time using the explicit Euler scheme to calculate the extrapolatation from past values

**Note:**
After a static step, it returns the value at the past step

#### 3.22.4.9 unsigned SPARK::TVariable::GetNumPastValues () const

Returns the number of past values that are being kept track of.

#### 3.22.4.10 double SPARK::TVariable::GetPastValue (unsigned *pastStep*) const [inline]

Returns the past value from idx steps ago.

#### 3.22.4.11 ]

double SPARK::TVariable::operator[ ] (unsigned *pastStep*) const [inline]

Returns the past value from idx steps ago.

#### 3.22.4.12 double SPARK::TVariable::GetInnerValue (unsigned *innerStep*) const [inline]

Access the intermediate values over the currrent step.

#### 3.22.4.13 double SPARK::TVariable::operator() (unsigned *innerStep*) const [inline]

Access the intermediate values over the currrent step.

#### 3.22.4.14 void SPARK::TVariable::SetFromLinkId (unsigned *var_id*) [inline]

Set INPUT_FROM_LINK=FromLink or PREDICT_FROM_LINK=FromLink.

#### 3.22.4.15 unsigned SPARK::TVariable::GetFromLinkId () const [inline]

Get var_id of FromLink variables, SPARK::NotAssigned if not specified.

#### 3.22.4.16 TObject* SPARK::TVariable::GetMatchedObject () [inline]

Returns the pointer to the object this variable is matched with

**Warning:**
If the variable is not an unknown, then there is no matched object and the data member is equal to 0.

#### 3.22.4.17 const TObject* SPARK::TVariable::GetMatchedObject () const [inline]

Returns the const pointer to the object this variable is matched with

**Warning:**
If the variable is not an unknown, then there is no matched object and the data member is equal to 0.

#### 3.22.4.18 unsigned SPARK::TVariable::GetHandle () const [inline]

Returns the unique handle as unsigned int.

#### 3.22.4.19 SPARK::VariableTypes SPARK::TVariable::GetType () const [inline]

Returns the variable type as SPARK::VariableTypes.

#### 3.22.4.20 unsigned long SPARK::TVariable::GetFlag () const [inline]

Returns the variable flag as an unsigned int.

#### 3.22.4.21 const std::string& SPARK::TVariable::GetName () const [inline]

Returns the variable name as a const char∗.

#### 3.22.4.22 const std::string& SPARK::TVariable::GetUnit () const [inline]

Returns the unit string as const char∗.

#### 3.22.4.23 void SPARK::TVariable::SetUnit (const char ∗ *unit*) [inline]

Sets the unit string to unit.

#### 3.22.4.24 double SPARK::TVariable::GetInit () const [inline]

Returns the initial value as double.

#### 3.22.4.25 void SPARK::TVariable::SetInit (double *init*) [inline]

Sets the initial value to init.

#### 3.22.4.26 double SPARK::TVariable::GetMin () const [inline]

Returns the mimimum value as double.

#### 3.22.4.27 void SPARK::TVariable::SetMin (double *min*) [inline]

Sets the minimum value to min.

Here is the call graph for this function:


#### 3.22.4.28 double SPARK::TVariable::GetMax () const [inline]

Returns the maximum value as double.

#### 3.22.4.29 void SPARK::TVariable::SetMax (double *max*) [inline]

Sets the maximum value to max.

Here is the call graph for this function:

#### 3.22.4.30 double SPARK::TVariable::GetAbsTolerance () const [inline]

Returns the absolute tolerance as double.

#### 3.22.4.31 void SPARK::TVariable::SetAbsTolerance (double *atol*)

Sets the absolute tolerance to atol.

#### 3.22.4.32 double SPARK::TVariable::GetScale () const [inline]

Returns the current scale.

#### 3.22.4.33 void SPARK::TVariable::SetScale (double *scalar*) [inline]

Sets the current scale to scalar.

#### 3.22.4.34 bool SPARK::TVariable::IsFlag (FlagTypes *flag*) const [inline]

Returns if flag is set for this variable, false otherwise.

#### 3.22.4.35 void SPARK::TVariable::Write (std::ostream & *os*, unsigned *width*, unsigned *precision*) const

Pretty printing of the variable name, its numerical value and its unit string to the output stream os

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

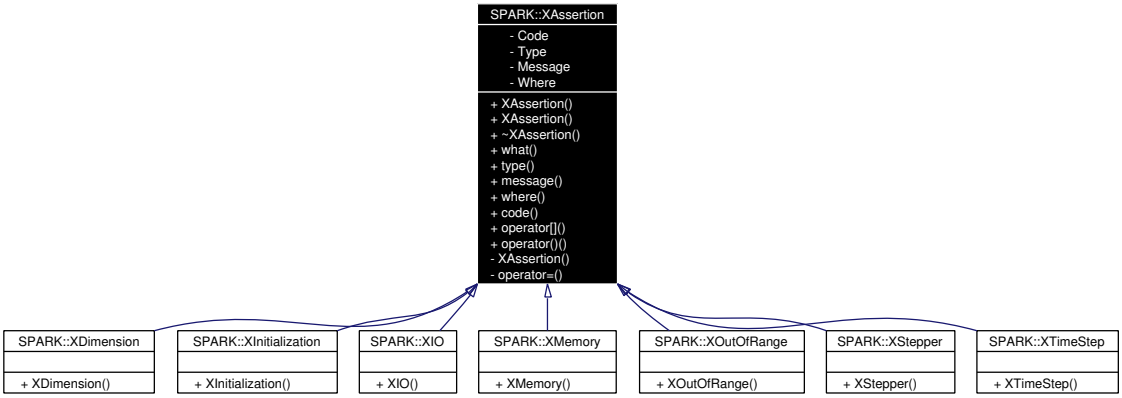
- variable.h

## 3.23 SPARK::XAssertion Class Reference

Base class for all SPARK exceptions.

`#include <exceptions.h>`

Inheritance diagram for SPARK::XAssertion:




### Public Types

- typedef enum ExitCodes TCode

  *Locally rename ExitCodes as TCode.*

### Public Member Functions

- XAssertion (const std::string &type, const std::string &where, const std::string &message, TCode code)

  *Constructor with message specified as std::string.*

- XAssertion (const XAssertion &x)

  *Copy constructor.*

- virtual ~XAssertion () throw ()

  *Trivial destructor.*

- virtual const char * what () const throw ()

  *Returns a description of the exception as a const char*.*

- const char * type () const throw ()

  *Returns type of the exception as const char*.*

- const char * message () const throw ()

  *Returns message associated with the exception as const char*.*

- const char * where () const throw ()

  *Returns where the exception was thrown from as const char*.*

- TCode code () const throw ()

*Returns code of the exception as TCode.*

- XAssertion & operator[ ] (const std::string &where)

  *Appends sender's information to Where string following " <- ".*

- XAssertion & operator() (const std::string &msg)

  *Appends msg to Message string on a new line.*

## 3.23.1 Detailed Description

Base class for all SPARK exceptions.

## 3.23.2 Member Typedef Documentation

### 3.23.2.1 typedef enum ExitCodes SPARK::XAssertion::TCode

Locally rename ExitCodes as TCode.

## 3.23.3 Constructor & Destructor Documentation

### 3.23.3.1 SPARK::XAssertion::XAssertion (const std::string & *type*, const std::string & *where*, const std::string & *message*, TCode *code*) [inline]

Constructor with message specified as std::string.

### 3.23.3.2 SPARK::XAssertion::XAssertion (const XAssertion & *x*) [inline]

Copy constructor.

### 3.23.3.3 virtual SPARK::XAssertion::~XAssertion () throw () [inline, virtual]

Trivial destructor.

## 3.23.4 Member Function Documentation

### 3.23.4.1 virtual const char* SPARK::XAssertion::what () const throw () [virtual]

Returns a description of the exception as a const char*.

### 3.23.4.2 const char* SPARK::XAssertion::type () const throw () [inline]

Returns type of the exception as const char*.

### 3.23.4.3 const char* SPARK::XAssertion::message () const throw () [inline]

Returns message associated with the exception as const char*.

#### 3.23.4.4 const char* SPARK::XAssertion::where () const throw () [inline]

Returns where the exception was thrown from as const char*.

#### 3.23.4.5 TCode SPARK::XAssertion::code () const throw () [inline]

Returns code of the exception as TCode.

#### 3.23.4.6 ]

XAssertion& SPARK::XAssertion::operator[ ] (const std::string & *where*)

Appends sender's information to Where string following " <- ".

#### 3.23.4.7 XAssertion& SPARK::XAssertion::operator() (const std::string & *msg*)

Appends msg to Message string on a new line.

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

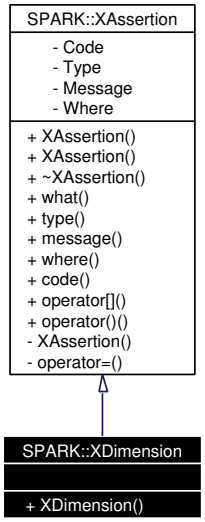
- exceptions.h

## 3.24 SPARK::XDimension Class Reference

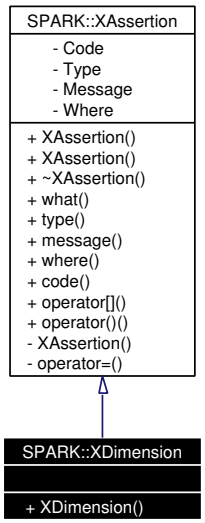
Indicates that a runtime error occured due to mismatched dimension.

`#include <exceptions.h>`

Inheritance diagram for SPARK::XDimension:




Collaboration diagram for SPARK::XDimension:




### Public Member Functions

- XDimension (const std::string &where, const std::string &message)

  *Constructor.*

### 3.24.1 Detailed Description

Indicates that a runtime error occured due to mismatched dimension.

### 3.24.2 Constructor & Destructor Documentation

#### 3.24.2.1 SPARK::XDimension::XDimension (const std::string & *where*, const std::string & *message*) [inline]

Constructor.

**Parameters:**

***where*** string specified as `const std::string&` indicating where in the source code this exception is thrown from

***message*** string specified as `const std::string&` documenting the reason for throwing this exception

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

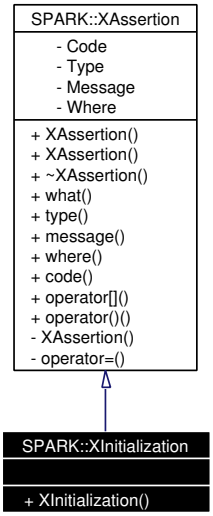
- exceptions.h

# 3.25 SPARK::XInitialization Class Reference

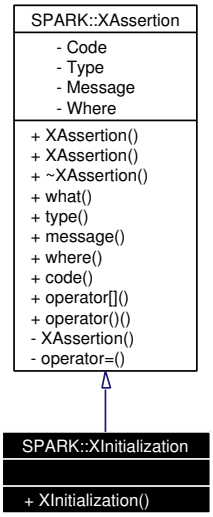
Indicates that a runtime error occured while initializing an object.

`#include <exceptions.h>`

Inheritance diagram for SPARK::XInitialization:




Collaboration diagram for SPARK::XInitialization:




## Public Member Functions

- XInitialization (const std::string &where, const std::string &message, TCode code)

  *Constructor.*

### 3.25.1 Detailed Description

Indicates that a runtime error occured while initializing an object.

### 3.25.2 Constructor & Destructor Documentation

#### 3.25.2.1 SPARK::XInitialization::XInitialization (const std::string & *where*, const std::string & *message*, TCode *code*) [inline]

Constructor.

**Parameters:**

***where*** string specified as `const std::string&` indicating where in the source code this exception is thrown from

***message*** string specified as `const std::string&` documenting the reason for throwing this exception

***code*** ExitCodes used to qualify the type of initilazation exception

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

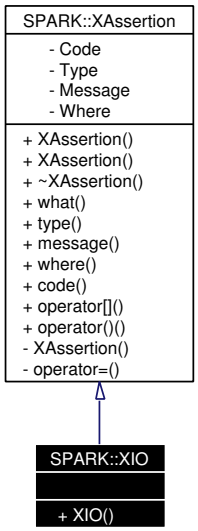
- exceptions.h

## 3.26 SPARK::XIO Class Reference

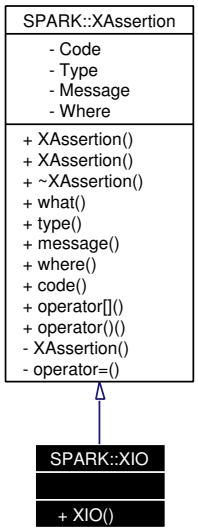
Indicates that a runtime error occured while performing an IO operation.

`#include <exceptions.h>`

Inheritance diagram for SPARK::XIO:




Collaboration diagram for SPARK::XIO:




### Public Member Functions

- XIO (const std::string &where, const std::string &message)

  *Constructor.*

### 3.26.1 Detailed Description

Indicates that a runtime error occured while performing an IO operation.

### 3.26.2 Constructor & Destructor Documentation

#### 3.26.2.1 SPARK::XIO::XIO (const std::string & *where*, const std::string & *message*) [inline]

Constructor.

**Parameters:**

***where*** string specified as `const std::string&` indicating where in the source code this exception is thrown from

***message*** string specified as `const std::string&` documenting the reason for throwing this exception

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

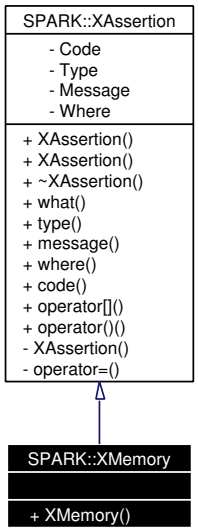
- exceptions.h

# 3.27 SPARK::XMemory Class Reference

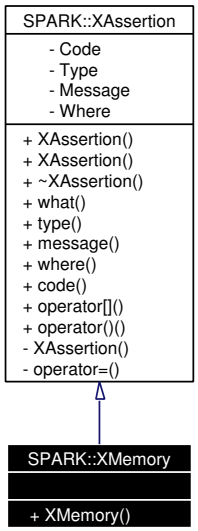
Indicates that a runtime error occured because memory could not be allocated.

`#include <exceptions.h>`

Inheritance diagram for SPARK::XMemory:




Collaboration diagram for SPARK::XMemory:




## Public Member Functions

- XMemory (const std::string &where, const std::string &message)

  *Constructor.*

### 3.27.1 Detailed Description

Indicates that a runtime error occured because memory could not be allocated.

### 3.27.2 Constructor & Destructor Documentation

#### 3.27.2.1 SPARK::XMemory::XMemory (const std::string & *where*, const std::string & *message*) [inline]

Constructor.

**Parameters:**

***where*** string specified as `const std::string&` indicating where in the source code this exception is thrown from

***message*** string specified as `const std::string&` documenting the reason for throwing this exception

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

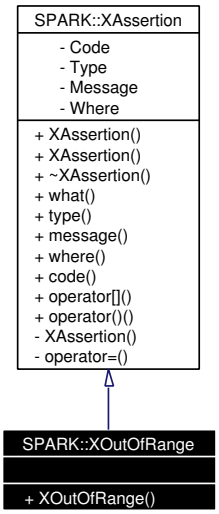
- exceptions.h

# 3.28 SPARK::XOutOfRange Class Reference

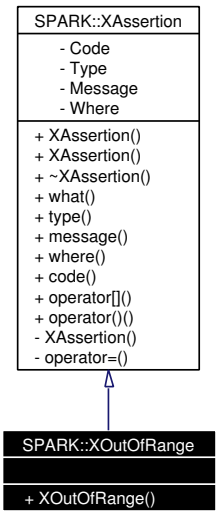
Indicates that a runtime error occured due to an out of range access operation on a container.

`#include <exceptions.h>`

Inheritance diagram for SPARK::XOutOfRange:




Collaboration diagram for SPARK::XOutOfRange:




## Public Member Functions

- XOutOfRange (const std::string &where, const std::string &message)

  *Constructor.*

### 3.28.1 Detailed Description

Indicates that a runtime error occured due to an out of range access operation on a container.

### 3.28.2 Constructor & Destructor Documentation

#### 3.28.2.1 SPARK::XOutOfRange::XOutOfRange (const std::string & *where*, const std::string & *message*) [inline]

Constructor.

**Parameters:**

***where*** string specified as `const std::string&` indicating where in the source code this exception is thrown from

***message*** string specified as `const std::string&` documenting the reason for throwing this exception

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

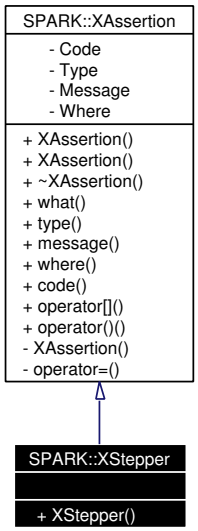
- exceptions.h

# 3.29 SPARK::XStepper Class Reference

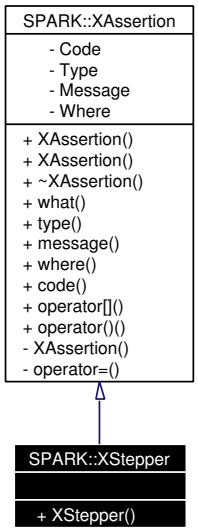
Inidicates that stepping to the next step failed.

`#include <exceptions.h>`

Inheritance diagram for SPARK::XStepper:




Collaboration diagram for SPARK::XStepper:




## Public Member Functions

- XStepper (const std::string &where, const std::string &message)

  *Constructor.*

### 3.29.1 Detailed Description

Inidicates that stepping to the next step failed.

### 3.29.2 Constructor & Destructor Documentation

#### 3.29.2.1 SPARK::XStepper::XStepper (const std::string & *where*, const std::string & *message*) [inline]

Constructor.

**Parameters:**

***where*** string specified as `const std::string&` indicating where in the source code this exception is thrown from

***message*** string specified as `const std::string&` documenting the reason for throwing this exception

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

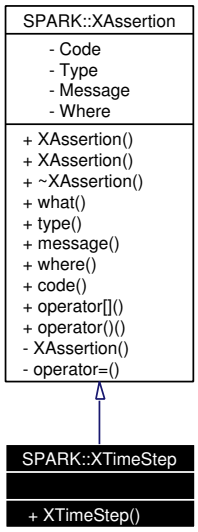
- exceptions.h

# 3.30 SPARK::XTimeStep Class Reference

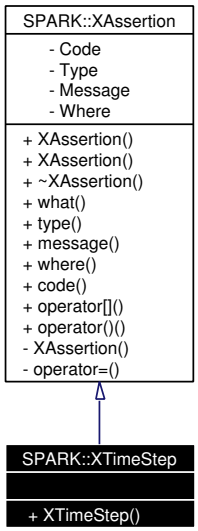
Indicates that a runtime error occured while adapting the time step.

`#include <exceptions.h>`

Inheritance diagram for SPARK::XTimeStep:




Collaboration diagram for SPARK::XTimeStep:




## Public Member Functions

- XTimeStep (const std::string &where, const std::string &message)

  *Constructor.*

### 3.30.1 Detailed Description

Indicates that a runtime error occured while adapting the time step.

### 3.30.2 Constructor & Destructor Documentation

#### 3.30.2.1 SPARK::XTimeStep::XTimeStep (const std::string & *where*, const std::string & *message*) [inline]

Constructor.

**Parameters:**

***where*** string specified as `const std::string&` indicating where in the source code this exception is thrown from

***message*** string specified as `const std::string&` documenting the reason for throwing this exception

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

- exceptions.h

# Chapter 4

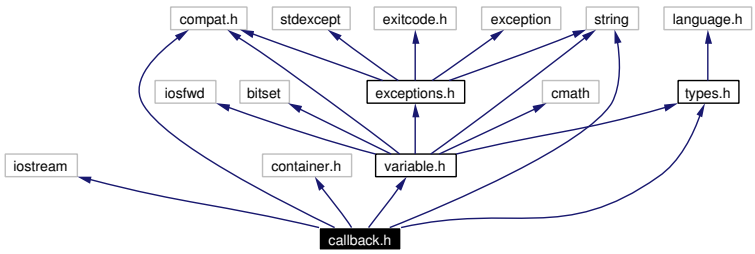
# SPARK Atomic Class API File Documentation

## 4.1 callback.h File Reference

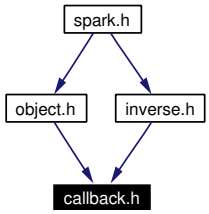
Declaration of the classes and typedefs used to implement the callback functions in the atomic classes.

```
#include <iostream>
#include <string>
#include "container.h"
#include "variable.h"
#include "types.h"
#include "compat.h"
```

Include dependency graph for callback.h:




This graph shows which files directly or indirectly include this file:



## Namespaces

- namespace SPARK

### 4.1.1 Detailed Description

Declaration of the classes and typedefs used to implement the callback functions in the atomic classes.

Types for the argument variables passed to the non-static callbacks:

- class SPARK::TArgument
- class SPARK::TArguments
- typedef SPARK::ArgList

Types for the target variables passed to the modifier callbacks:

- class SPARK::TTarget
- class SPARK::TTargets
- typedef SPARK::TargetList

Callback wrapper classes:

- template<typename FreeFuncPtrType> class SPARK::TFreeFunctionWrapper
- class SPARK::TModifierCallback
- class SPARK::TNonModifierCallback
- class SPARK::TPredicateCallback
- class SPARK::TStaticNonModifierCallback
- class SPARK::TStaticPredicateCallback

**Attention:**



**Author:**

Dimitri Curtil (LBNL/SRG)

**Date:**

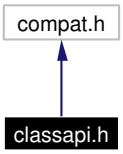
December 20, 2002

## 4.2 classapi.h File Reference

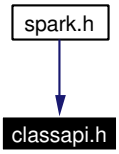
Header file that defines templatized utility functions used to implement the private data management inside the atomic classes.

`#include "compat.h"`

Include dependency graph for classapi.h:




This graph shows which files directly or indirectly include this file:




### Namespaces

- namespace SPARK::AtomicClass
- namespace SPARK

### 4.2.1 Detailed Description

Header file that defines templatized utility functions used to implement the private data management inside the atomic classes.

Functions to:

- set the private data
- retrieve the private data
- deallocate the private data

Each utility function comes in 2 flavors:

- for the SPARK::TObject instance as accessed from an instance callback
- for the SPARK::TInverse instance as accessed from a static callback

Also, the DeleteData() functions are parameterized with a delete policy type that lets you implement the deletion task. We have provided two such policy classes:

- SPARK::delete_policy
- SPARK::delete_array_policy

**Author:**
Dimitri Curtil

**Version:**
SPARK 2.0

**Date:**
April 16, 2003

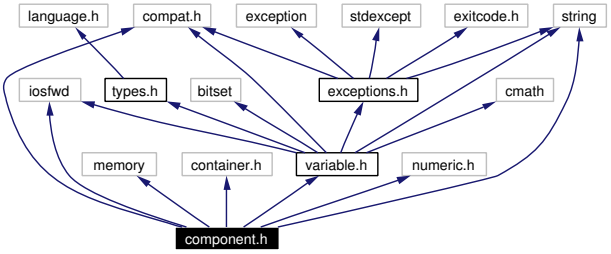
**Attention:**

## 4.3 component.h File Reference

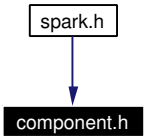
Declaration of the SPARK::TComponent class.

```
#include <iosfwd>
#include <string>
#include <memory>
#include "container.h"
#include "compat.h"
#include "numeric.h"
#include "variable.h"
```

Include dependency graph for component.h:




This graph shows which files directly or indirectly include this file:




### Namespaces

- namespace SPARK

### 4.3.1 Detailed Description

Declaration of the SPARK::TComponent class.

**Author:**
Dimitri Curtil (LBNL/SRG)

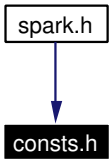
**Date:**
September 9, 2002

**Attention:**

## 4.4 consts.h File Reference

Declaration of constants used in the HVAC toolkit.

This graph shows which files directly or indirectly include this file:




### Mathematical constants

- const double SMALL = 1.0e-23
  *considered a very small number*
- const double LARGE = 1.0e+23
  *considered a very large number*
- const double PI = 3.14159265358979323846
  *ratio of circle perimeter over its diameter: $\pi$*

### Constants used to identify flow arrangement in heat exchangers

- const unsigned COUNTER_FLOW = 1
  *counter flow*
- const unsigned PARALLEL_FLOW = 2
  *parallel flow*
- const unsigned CROSS_FLOW_BOTH_UNMIXED = 3
  *cross flow, both unmixed*
- const unsigned CROSS_FLOW_BOTH_MIXED = 4
  *cross flow, both mixed*
- const unsigned CROSS_FLOW_1_UNMIXED = 5
  *cross flow, first stream unmixed*
- const unsigned CROSS_FLOW_2_UNMIXED = 6
  *cross flow, second stream unmixed*

### Physical constants

- const double ABS_ZERO = -273.16
  *absolute zero temperature in [deg_C]*

- const double KELV_ZERO = 273.16
  *0 [deg_C] in [K]*
- const double BOLTZ = 5.67E-8
  *Stefan-Boltzmann constant [W/(m^2∗K^4)].*
- const double CP_AIR = 1006.0
  *specific heat capacity of dry air [J/(kg∗K)]*
- const double MW_AIR = 28.9645
  *molar weight of dry air [g/mol]*
- const double CP_VAP = 1805.0
  *specific heat capacity of water vapor [J/(kg∗K)]*
- const double CP_WAT = 4186.0
  *specific heat capacity of liquid water [J/(kg∗K)]*
- const double MW_WATER = 18.01528
  *molar weight of liquid water [g/mol]*
- const double MW_RATIO = 0.62197
  *ratio of molar weights of liquid water over dry air [-]*
- const double HF_VAP = 2.501E6
  *latent heat of vaporization of water [J/kg]*
- const double LAMBDA_AIR = 0.0243
  *thermal conductivity of dry air [W/(m∗K)]*
- const double LAMBDA_WAT = 0.554
  *thermal conductivity of liquid water [W/(m∗K)]*
- const double PRANDTL_AIR = 0.71
  *Prandtl number for dry air [-].*
- const double RHO_AIR = 1.2
  *density of air [kg/m^3]*
- const double RHO_WAT = 998.0
  *density of water [kg/m^3]*
- const double VISC_WAT = 1.0E-3
  *dynamic viscosity for liquid water [kg/(m∗s)]*
- const double R_AIR = 287.053
  *specific gas constant for ideal air [J/(kg∗K)]*
- const double R_0 = 8314.34
  *universal gas constant [kJ/(mol∗K)]*

- const double P_ATM = 101325.0

  *atmospheric pressure in [kg/(m∗s^2)] or [Pa]*

### 4.4.1 Detailed Description

Declaration of constants used in the HVAC toolkit.

**Attention:**



### 4.4.2 Variable Documentation

#### 4.4.2.1 const double SMALL = 1.0e-23

considered a very small number

#### 4.4.2.2 const double LARGE = 1.0e+23

considered a very large number

#### 4.4.2.3 const double PI = 3.14159265358979323846

ratio of circle perimeter over its diameter: $\pi$

#### 4.4.2.4 const unsigned COUNTER_FLOW = 1

counter flow

#### 4.4.2.5 const unsigned PARALLEL_FLOW = 2

parallel flow

#### 4.4.2.6 const unsigned CROSS_FLOW_BOTH_UNMIXED = 3

cross flow, both unmixed

#### 4.4.2.7 const unsigned CROSS_FLOW_BOTH_MIXED = 4

cross flow, both mixed

#### 4.4.2.8 const unsigned CROSS_FLOW_1_UNMIXED = 5

cross flow, first stream unmixed

#### 4.4.2.9 const unsigned CROSS_FLOW_2_UNMIXED = 6

cross flow, second stream unmixed

#### 4.4.2.10 const double ABS_ZERO = -273.16

absolute zero temperature in [deg_C]

#### 4.4.2.11 const double KELV_ZERO = 273.16

0 [deg_C] in [K]

#### 4.4.2.12 const double BOLTZ = 5.67E-8

Stefan-Boltzmann constant [W/(m^2∗K^4)].

#### 4.4.2.13 const double CP_AIR = 1006.0

specific heat capacity of dry air [J/(kg∗K)]

#### 4.4.2.14 const double MW_AIR = 28.9645

molar weight of dry air [g/mol]

#### 4.4.2.15 const double CP_VAP = 1805.0

specific heat capacity of water vapor [J/(kg∗K)]

#### 4.4.2.16 const double CP_WAT = 4186.0

specific heat capacity of liquid water [J/(kg∗K)]

#### 4.4.2.17 const double MW_WATER = 18.01528

molar weight of liquid water [g/mol]

#### 4.4.2.18 const double MW_RATIO = 0.62197

ratio of molar weights of liquid water over dry air [-]

#### 4.4.2.19 const double HF_VAP = 2.501E6

latent heat of vaporization of water [J/kg]

#### 4.4.2.20 const double LAMBDA_AIR = 0.0243

thermal conductivity of dry air [W/(m∗K)]

#### 4.4.2.21 const double LAMBDA_WAT = 0.554

thermal conductivity of liquid water [W/(m∗K)]

#### 4.4.2.22 const double PRANDTL_AIR = 0.71

Prandtl number for dry air [-].

#### 4.4.2.23 const double RHO_AIR = 1.2

density of air [kg/m^3]

#### 4.4.2.24 const double RHO_WAT = 998.0

density of water [kg/m^3]

#### 4.4.2.25 const double VISC_WAT = 1.0E-3

dynamic viscosity for liquid water [kg/(m∗s)]

#### 4.4.2.26 const double R_AIR = 287.053

specific gas constant for ideal air [J/(kg∗K)]

#### 4.4.2.27 const double R_0 = 8314.34

universal gas constant [kJ/(mol∗K)]

#### 4.4.2.28 const double P_ATM = 101325.0

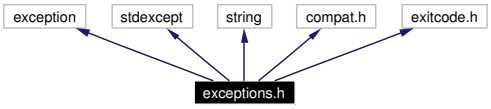
atmospheric pressure in [kg/(m∗s^2)] or [Pa]

## 4.5 exceptions.h File Reference

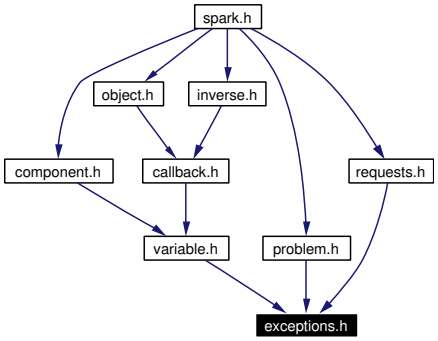
Declaration of the generic exception classes used in the SPARK solver.

```
#include <exception>
#include <stdexcept>
#include <string>
#include "compat.h"
#include "exitcode.h"
```

Include dependency graph for exceptions.h:




This graph shows which files directly or indirectly include this file:




### Namespaces

- namespace SPARK

### 4.5.1 Detailed Description

Declaration of the generic exception classes used in the SPARK solver.

- class SPARK::XAssertion
- class SPARK::XDimension
- class SPARK::XOutOfRange
- class SPARK::XMemory
- class SPARK::XInitialization
- class SPARK::XIO
- class SPARK::XTimeStep

- class SPARK::XStepper

**Attention:**



**Author:**

Dimitri Curtil (LBNL/SRG)

**Date:**

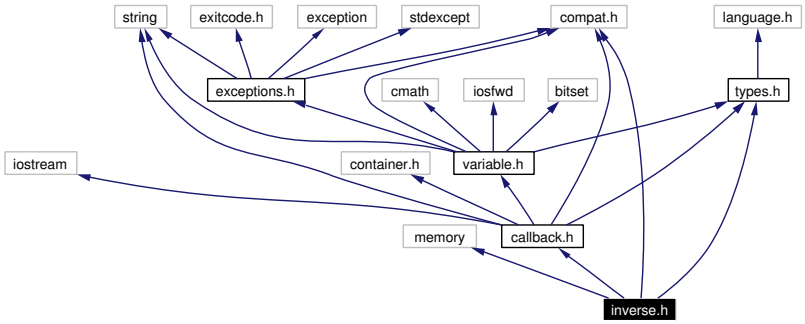
September 5, 2002

# 4.6 inverse.h File Reference

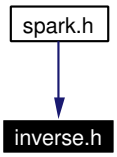
Declaration of the SPARK::TInverse class.

```
#include <memory>
#include "callback.h"
#include "types.h"
#include "compat.h"
```

Include dependency graph for inverse.h:




This graph shows which files directly or indirectly include this file:




## Namespaces

- namespace SPARK

### 4.6.1 Detailed Description

Declaration of the SPARK::TInverse class.

**Attention:**

PORTIONS COPYRIGHT (C) 2003 AYRES SOWELL ASSOCIATES, INC.
PORTIONS COPYRIGHT (C) 2003 THE REGENTS OF THE UNIVERSITY OF CALIFORNIA . PENDING APPROVAL BY THE US DEPARTMENT OF ENERGY. ALL RIGHTS RESERVED.

**Author:**

Dimitri Curtil (LBNL/SRG)

**Date:**

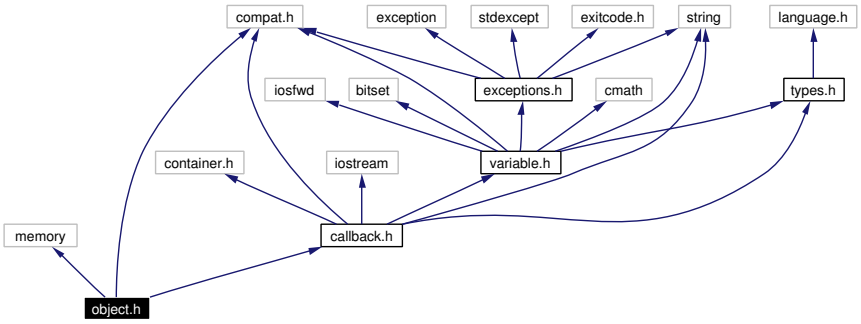
January 10, 2003

## 4.7 object.h File Reference

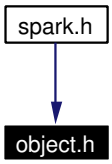
Declaration of the SPARK::TObject class.

```
#include <memory>
#include "callback.h"
#include "compat.h"
```

Include dependency graph for object.h:




This graph shows which files directly or indirectly include this file:




### Namespaces

- namespace SPARK

### 4.7.1 Detailed Description

Declaration of the SPARK::TObject class.

**Attention:**

PORTIONS COPYRIGHT (C) 2003 AYRES SOWELL ASSOCIATES, INC.
PORTIONS COPYRIGHT (C) 2003 THE REGENTS OF THE UNIVERSITY OF CALIFORNIA . PENDING APPROVAL BY THE US DEPARTMENT OF ENERGY. ALL RIGHTS RESERVED.

**Author:**

Dimitri Curtil (LBNL/SRG)

**Date:**

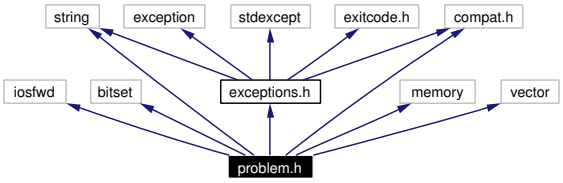
January 12, 2003

# 4.8 problem.h File Reference

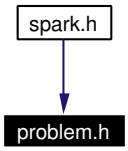
Declaration of the classes SPARK::TProblem and SPARK::TProblem::TState.

```
#include <iosfwd>
#include <bitset>
#include <string>
#include <memory>
#include <vector>
#include "compat.h"
#include "exceptions.h"
```

Include dependency graph for problem.h:




This graph shows which files directly or indirectly include this file:




## Namespaces

- namespace SPARK
- namespace SPARK::Requests

## Defines

- #define __PROBLEM_H__

### 4.8.1 Detailed Description

Declaration of the classes SPARK::TProblem and SPARK::TProblem::TState.

### 4.8.2 Define Documentation

#### 4.8.2.1 #define __PROBLEM_H__

**Attention:**
PORTIONS COPYRIGHT (C) 2003 AYRES SOWELL ASSOCIATES, INC.

**Author:**
Dimitri Curtil (LBNL/SRG)

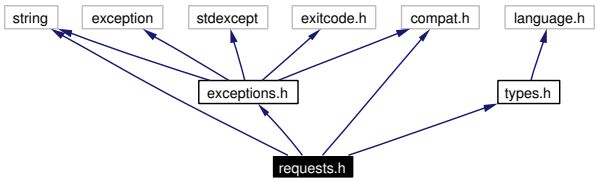
**Date:**
April 20, 2001

# 4.9 requests.h File Reference

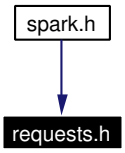
Declaration of the request classes accessible from the atomic classes.

```
#include <string>
#include "types.h"
#include "compat.h"
#include "exceptions.h"
```

Include dependency graph for requests.h:




This graph shows which files directly or indirectly include this file:




## Namespaces

- namespace SPARK
- namespace SPARK::Requests

### 4.9.1 Detailed Description

Declaration of the request classes accessible from the atomic classes.

**Attention:**

PORTIONS COPYRIGHT (C) 2003 AYRES SOWELL ASSOCIATES, INC.
PORTIONS COPYRIGHT (C) 2003 THE REGENTS OF THE UNIVERSITY OF CALIFORNIA . PENDING APPROVAL BY THE US DEPARTMENT OF ENERGY. ALL RIGHTS RESERVED.

**Author:**

Dimitri Curtil (LBNL/SRG)

**Date:**

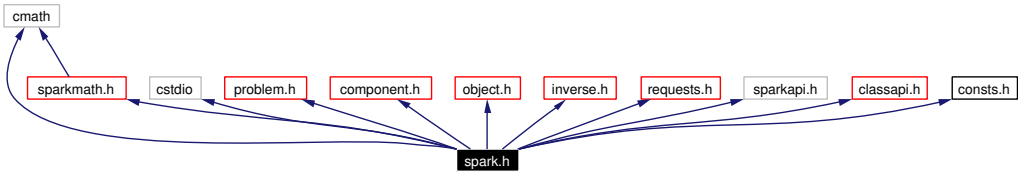
February 10, 2003

## 4.10 spark.h File Reference

Header file required to define SPARK inverses.

```
#include <cmath>
#include <cstdio>
#include "problem.h"
#include "component.h"
#include "object.h"
#include "inverse.h"
#include "requests.h"
#include "sparkmath.h"
#include "sparkapi.h"
#include "classapi.h"
#include "consts.h"
```

Include dependency graph for spark.h:




### Preprocessor macro definitions used in the implementation of the various callbacks in the atomic classes

**Note:**

The callback prototypes defined for each callback type must match the typedef's in the file callback.h

- #define **CALLBACK_DECLSPEC**(callback_type) extern "C" callback_type
- #define NON_MODIFIER_CALLBACK(func) CALLBACK_DECLSPEC(void) func(SPARK::TObject* this_pointer, SPARK::ArgList args)

  *Macros to define the prototype for a modifier callback.*

- #define MODIFIER_CALLBACK(func) CALLBACK_DECLSPEC(void) func(SPARK::TObject* this_pointer, SPARK::ArgList args, SPARK::TargetList targets)

  *Macros to define the prototype for a non-modifier callback.*

- #define PREDICATE_CALLBACK(func) CALLBACK_DECLSPEC(bool) func(SPARK::TObject* this_pointer, SPARK::ArgList args)

  *Macros to define the prototype for a predicate callback.*

- #define STATIC_NON_MODIFIER_CALLBACK(func) CALLBACK_DECLSPEC(void) func(SPARK::TInverse* this_-pointer)

  *Macros to define the prototype for a static callback.*

- #define STATIC_PREDICATE_CALLBACK(func) CALLBACK_DECLSPEC(bool) func(SPARK::TInverse* this_pointer)

  *Macros to define the prototype for a static predicate callback.*

- #define EVALUATE(func) MODIFIER_CALLBACK( func )

  *Macros to declare an EVALUATE callback.*

- #define PREDICT(func) MODIFIER_CALLBACK( func )

  *Macros to declare a PREDICT callback.*

- #define CONSTRUCT(func) NON_MODIFIER_CALLBACK( func )

  *Macros to declare a CONSTRUCT callback.*

- #define PREPARE_STEP(func) NON_MODIFIER_CALLBACK( func )

  *Macros to declare a PREPARE_STEP callback.*

- #define CHECK_INTEGRATION_STEP(func) PREDICATE_CALLBACK( func )

  *Macros to declare a CHECK_INTEGRATION_STEP callback.*

- #define COMMIT(func) NON_MODIFIER_CALLBACK( func )

  *Macros to declare a COMMIT callback.*

- #define ROLLBACK(func) NON_MODIFIER_CALLBACK( func )

  *Macros to declare a ROLLBACK callback.*

- #define DESTRUCT(func) NON_MODIFIER_CALLBACK( func )

  *Macros to declare a DESTRUCT callback.*

- #define STATIC_CONSTRUCT(func) STATIC_NON_MODIFIER_CALLBACK( func )

  *Macros to declare a STATIC_CONSTRUCT callback.*

- #define STATIC_PREPARE_STEP(func) STATIC_NON_MODIFIER_CALLBACK( func )

  *Macros to declare a STATIC_PREPARE_STEP callback.*

- #define STATIC_CHECK_INTEGRATION_STEP(func) STATIC_PREDICATE_CALLBACK( func )

  *Macros to declare a STATIC_CHECK_INTEGRATION_STEP callback.*

- #define STATIC_COMMIT(func) STATIC_NON_MODIFIER_CALLBACK( func )

  *Macros to declare a STATIC_COMMIT callback.*

- #define STATIC_ROLLBACK(func) STATIC_NON_MODIFIER_CALLBACK( func )

  *Macros to declare a STATIC_ROLLBACK callback.*

- #define STATIC_DESTRUCT(func) STATIC_NON_MODIFIER_CALLBACK( func )

  *Macros to declare a STATIC_DESTRUCT callback.*

- #define TARGET(pos, name) SPARK::TTarget& name = targets[pos]

  *A macro that defines a reference named "name" to the TTarget object stored in position "pos" of the target list.*

- #define ARGUMENT(pos, name) const SPARK::TArgument& name = args[pos]

  *A macro that defines a const reference named "name" to the TArgument object of the argument stored in position "pos".*

- #define ARGDEF(pos, name) ARGUMENT( pos, name )

  *A macro that defines a const reference named "name" to the TArgument object of the argument stored in position "pos".*

- #define RETURN(val) { targets[0] = val; return; }

  *A macro that returns the scalar val for a single-valued inverse.*

- #define ACCEPT_STEP true

  *Macros used in predicate callbacks to indicate that the step is accepted.*

- #define REJECT_STEP false

  *Macros used in predicate callbacks to indicate that the step is rejected.*

## Preprocessor macros to access active context from within a callback

- #define THIS this_pointer

  *A macro to access the pointer to the current "this" pointer associated with the current callback.*

- #define GET_DATA(owner, concrete_type, name) concrete_type* name = SPARK::AtomicClass::GetData<concrete_type>( owner )

  *A macro to access the pointer to the private data object stored as part of an object.*

- #define SET_DATA(owner, concrete_type, name) SPARK::AtomicClass::SetData<concrete_type>( owner, name )

  *A macro to store the pointer to the private data stored as part of an object.*

- #define DELETE_DATA(owner, concrete_type, is_array_flag) SPARK::AtomicClass::DeleteData<concrete_type, SPARK::deleter<is_array_flag> >( owner )

  *A macro to delete data pointed to by an object.*

- #define ACTIVE_PROBLEM THIS → GetProblem()

  *Returns pointer to TProblem object this object belongs to.*

- #define ACTIVE_INVERSE THIS → GetInverse()

  *Returns pointer to TInverse object this object belongs to.*

- #define ACTIVE_COMPONENT THIS → GetComponent()

  *Returns pointer to TComponent object this object belongs to.*

- #define ERROR_LOG(msg) SPARK::Log( SPARK::GetErrorLog(), THIS, __LINE__, msg )

  *Writes msg specified as const char* to the error log file.*

- #define RUN_LOG(msg) SPARK::Log( SPARK::GetRunLog(), THIS, __LINE__, msg )

  *Writes msg specified as const char* to the run log file.*

## Preprocessor macros to send a request to the active problem from within a callback

- #define REQUEST__HEADER(sender, target, context) SPARK::Requests::THeader( (sender), (target), (context) )

  *Macro to generate a request header with specific context information as* `const char*`.

- #define REQUEST__ABORT(context) SPARK::Requests::TDispatcher::abort( REQUEST__HEADER( THIS, ACTIVE_-PROBLEM, context ) )

  *Macro to send an abort request from an atomic class.*

- #define REQUEST__STOP(context) SPARK::Requests::TDispatcher::stop( REQUEST__HEADER( THIS, ACTIVE_-PROBLEM, context ) )

  *Macro to send a stop request from an atomic class.*

- #define REQUEST__SET_STOP_TIME(context, time) SPARK::Requests::TDispatcher::set_stop_time( REQUEST__-HEADER( THIS, ACTIVE_PROBLEM, context ), time )

  *Macro to send a set stop time request from an atomic class.*

- #define REQUEST__REPORT(context) SPARK::Requests::TDispatcher::report( REQUEST__HEADER( THIS, ACTIVE_-PROBLEM, context ) )

  *Macro to send a report request from an atomic class.*

- #define REQUEST__SNAPSHOT(context, filename) SPARK::Requests::TDispatcher::snapshot( REQUEST__HEADER( THIS, ACTIVE_PROBLEM, context ), filename )

  *Macro to send a snapshot request from an atomic class.*

- #define REQUEST__SET_MEETING_POINT(context, time) SPARK::Requests::TDispatcher::set_meeting_point( REQUEST__HEADER( THIS, ACTIVE_PROBLEM, context ), time )

  *Macro to send a request to synchronize with a meeting point from an atomic class.*

- #define REQUEST__CLEAR_MEETING_POINTS(context) SPARK::Requests::TDispatcher::clear_meeting_points( REQUEST__HEADER( THIS, ACTIVE_PROBLEM, context ) )

  *Macro to send a clear meeting points request from an atomic class.*

- #define REQUEST__RESTART(context) SPARK::Requests::TDispatcher::restart( REQUEST__HEADER( THIS, ACTIVE_PROBLEM, context ) )

  *Macro to send a restart request from an atomic class.*

- #define REQUEST__SET_DYNAMIC_STEPPER(context, stepper) SPARK::Requests::TDispatcher::set_dynamic_stepper( REQUEST__HEADER( THIS, ACTIVE_PROBLEM, context ), stepper )

  *Macro to send a "set dynamic stepper" request from an atomic class.*

- #define REQUEST__SET_TIME_STEP(context, h) SPARK::Requests::TDispatcher::set_time_step( REQUEST__-HEADER( THIS, ACTIVE_PROBLEM, context ), h )

  *Macro to send a "set time step" request from an atomic class.*

### 4.10.1 Detailed Description

Header file required to define SPARK inverses.

This header file should be included in the file where the inverses of an atomic class are implemented :

- to define the preprocessor macros.
- to include the required header files.

**Author:**
Dimitri Curtil

**Version:**
SPARK 2.0

**Date:**
September 3, 2003 Added SET_DATA, GET_DATA and DELETE_DATA macros
December 18, 2002 Original implementation

**Attention:**


## 4.10.2 Define Documentation

### 4.10.2.1 #define NON_MODIFIER_CALLBACK(func) CALLBACK_DECLSPEC(void) func(SPARK::TObject* this_pointer, SPARK::ArgList args)

Macros to define the prototype for a modifier callback.

### 4.10.2.2 #define MODIFIER_CALLBACK(func) CALLBACK_DECLSPEC(void) func(SPARK::TObject* this_pointer, SPARK::ArgList args, SPARK::TargetList targets)

Macros to define the prototype for a non-modifier callback.

### 4.10.2.3 #define PREDICATE_CALLBACK(func) CALLBACK_DECLSPEC(bool) func(SPARK::TObject* this_pointer, SPARK::ArgList args)

Macros to define the prototype for a predicate callback.

### 4.10.2.4 #define STATIC_NON_MODIFIER_CALLBACK(func) CALLBACK_DECLSPEC(void) func(SPARK::TInverse* this_pointer)

Macros to define the prototype for a static callback.

### 4.10.2.5 #define STATIC_PREDICATE_CALLBACK(func) CALLBACK_DECLSPEC(bool) func(SPARK::TInverse* this_pointer)

Macros to define the prototype for a static predicate callback.

### 4.10.2.6 #define EVALUATE(func) MODIFIER_CALLBACK( func )

Macros to declare an EVALUATE callback.

**Examples:**
analytical_frst_ord.cc, analytical_spring.cc, integrator_euler.cc, and sum.cc.

### 4.10.2.7 #define PREDICT(func) MODIFIER_CALLBACK( func )

Macros to declare a PREDICT callback.

**Examples:**
integrator_euler.cc.

#### 4.10.2.8 #define CONSTRUCT(func) NON_MODIFIER_CALLBACK( func )

Macros to declare a CONSTRUCT callback.

**Examples:**
analytical_frst_ord.cc, analytical_spring.cc, and integrator_euler.cc.

#### 4.10.2.9 #define PREPARE_STEP(func) NON_MODIFIER_CALLBACK( func )

Macros to declare a PREPARE_STEP callback.

**Examples:**
integrator_euler.cc.

#### 4.10.2.10 #define CHECK_INTEGRATION_STEP(func) PREDICATE_CALLBACK( func )

Macros to declare a CHECK_INTEGRATION_STEP callback.

**Examples:**
integrator_euler.cc.

#### 4.10.2.11 #define COMMIT(func) NON_MODIFIER_CALLBACK( func )

Macros to declare a COMMIT callback.

**Examples:**
integrator_euler.cc.

#### 4.10.2.12 #define ROLLBACK(func) NON_MODIFIER_CALLBACK( func )

Macros to declare a ROLLBACK callback.

**Examples:**
integrator_euler.cc.

#### 4.10.2.13 #define DESTRUCT(func) NON_MODIFIER_CALLBACK( func )

Macros to declare a DESTRUCT callback.

**Examples:**
analytical_frst_ord.cc, analytical_spring.cc, and integrator_euler.cc.

#### 4.10.2.14 #define STATIC_CONSTRUCT(func) STATIC_NON_MODIFIER_CALLBACK( func )

Macros to declare a STATIC_CONSTRUCT callback.

**Examples:**
integrator_euler.cc.

#### 4.10.2.15 #define STATIC_PREPARE_STEP(func) STATIC_NON_MODIFIER_CALLBACK( func )

Macros to declare a STATIC_PREPARE_STEP callback.

**Examples:**
integrator_euler.cc.

#### 4.10.2.16 #define STATIC_CHECK_INTEGRATION_STEP(func) STATIC_PREDICATE_CALLBACK( func )

Macros to declare a STATIC_CHECK_INTEGRATION_STEP callback.

**Examples:**
integrator_euler.cc.

#### 4.10.2.17 #define STATIC_COMMIT(func) STATIC_NON_MODIFIER_CALLBACK( func )

Macros to declare a STATIC_COMMIT callback.

**Examples:**
integrator_euler.cc.

#### 4.10.2.18 #define STATIC_ROLLBACK(func) STATIC_NON_MODIFIER_CALLBACK( func )

Macros to declare a STATIC_ROLLBACK callback.

**Examples:**
integrator_euler.cc.

#### 4.10.2.19 #define STATIC_DESTRUCT(func) STATIC_NON_MODIFIER_CALLBACK( func )

Macros to declare a STATIC_DESTRUCT callback.

**Examples:**
integrator_euler.cc.

#### 4.10.2.20 #define TARGET(pos, name) SPARK::TTarget& name = targets[pos]

A macro that defines a reference named "name" to the TTarget object stored in position "pos" of the target list.

**Examples:**
analytical_frst_ord.cc, analytical_spring.cc, and integrator_euler.cc.

#### 4.10.2.21 #define ARGUMENT(pos, name) const SPARK::TArgument& name = args[pos]

A macro that defines a const reference named "name" to the TArgument object of the argument stored in position "pos".

**Examples:**
analytical_frst_ord.cc, analytical_spring.cc, and integrator_euler.cc.

#### 4.10.2.22 #define ARGDEF(pos, name) ARGUMENT( pos, name )

A macro that defines a const reference named "name" to the TArgument object of the argument stored in position "pos".

**Note:**
This macro is defined for backward-compatibility reasons since it was used in the HVAC toolkit and globalclass implementations of SPARK 1.xx. The new macro ARGUMENT should be preferred to this one.

**Examples:**
sum.cc.

#### 4.10.2.23 #define RETURN(val) { targets[0] = val; return; }

A macro that returns the scalar val for a single-valued inverse.

**Note:**
There is no equivalent macro for multivalued inverses. The target value must be explicitly assigned to each declared TTarget instances.

**Examples:**
sum.cc.

#### 4.10.2.24 #define ACCEPT_STEP true

Macros used in predicate callbacks to indicate that the step is accepted.

#### 4.10.2.25 #define REJECT_STEP false

Macros used in predicate callbacks to indicate that the step is rejected.

#### 4.10.2.26 #define THIS this_pointer

A macro to access the pointer to the current "this" pointer associated with the current callback.

**Note:**
In an instance callback the private data object is implemented as a pointer to a TObject instance
In a static callback the private data object is implemented as a pointer to a TInverse instance

#### 4.10.2.27 #define GET_DATA(owner, concrete_type, name) concrete_type* name = SPARK::AtomicClass::GetData<concrete_type>( owner )

A macro to access the pointer to the private data object stored as part of an object.

**Parameters:**
***owner*** Either SPARK::TInverse* or SPARK::TObject* owner of the data

***concrete_type*** Concrete type of the data stored in owner used for type-casting

***name*** Name of the variable pointing to the data of type `concrete_type` to be stored in owner

**Note:**
The macro implements a static_cast from void* to the specified `concrete_type`.

**Examples:**
analytical_frst_ord.cc, and integrator_euler.cc.

#### 4.10.2.28 #define SET_DATA(owner, concrete_type, name) SPARK::AtomicClass::SetData<concrete_type>( owner, name )

A macro to store the pointer to the private data stored as part of an object.

**Parameters:**
***owner*** Either SPARK::TInverse* or SPARK::TObject* owner of the data

***concrete_type*** Concrete type of the data stored in owner used for type-casting

***name*** Name of the variable pointing to the data of type `concrete_type` to be stored in owner

**Examples:**
analytical_frst_ord.cc, and integrator_euler.cc.

#### 4.10.2.29 #define DELETE_DATA(owner, concrete_type, is_array_flag) SPARK::AtomicClass::DeleteData<concrete_type, SPARK::deleter<is_array_flag> >( owner )

A macro to delete data pointed to by an object.

**Parameters:**
***owner*** Either SPARK::TInverse* or SPARK::TObject* owner of the data

***concrete_type*** Concrete type of the data stored in owner used for type-casting

***is_array_flag*** Boolean value (true or false) indicating whether the data is a built-in array or not

**Warning:**
When deleting a built-in array, make sure to specify isArray == true to call the delete [] operator. Otherwise specify isArray = false for non "built-in array" types.

**Examples:**
analytical_frst_ord.cc, and integrator_euler.cc.

#### 4.10.2.30 #define ACTIVE_PROBLEM THIS → GetProblem()

Returns pointer to TProblem object this object belongs to.

**Note:**
This applies to both static and non-static callbacks.

#### 4.10.2.31 #define ACTIVE_INVERSE THIS → GetInverse()

Returns pointer to TInverse object this object belongs to.

**Warning:**
This applies only to instance callbacks.

#### 4.10.2.32 #define ACTIVE_COMPONENT THIS → GetComponent()

Returns pointer to TComponent object this object belongs to.

**Warning:**
This applies only to instance callbacks.

#### 4.10.2.33 #define ERROR_LOG(msg) SPARK::Log( SPARK::GetErrorLog(), THIS, __LINE__, msg )

Writes msg specified as const char* to the error log file.

**Note:**
Users could also use the API function SPARK::GetErrorLog() to retrieve a reference to the std::ostream object for the error log file.

#### 4.10.2.34 #define RUN_LOG(msg) SPARK::Log( SPARK::GetRunLog(), THIS, __LINE__, msg )

Writes msg specified as const char* to the run log file.

**Note:**
Users could also use the API function SPARK::GetRunLog() to retrieve a reference to the std::ostream object for the run log file.

**Examples:**
analytical_frst_ord.cc, and analytical_spring.cc.

#### 4.10.2.35 #define REQUEST__HEADER(sender, target, context) SPARK::Requests::THeader( (sender), (target), (context) )

Macro to generate a request header with specific context information as `const char*`.

**Parameters:**
***sender*** is a TInverse* or TObject* or TProblem*

***target*** is a TProblem*

***context*** is a `const char*` object that contains the description of the calling context

#### 4.10.2.36 #define REQUEST__ABORT(context) SPARK::Requests::TDispatcher::abort( REQUEST__HEADER( THIS, ACTIVE_PROBLEM, context ) )

Macro to send an abort request from an atomic class.

See SPARK::Requests::TDispatcher::abort()

**Parameters:**
***context*** is a `const char*` that contains the description of the calling context

**Examples:**
analytical_frst_ord.cc, analytical_spring.cc, and integrator_euler.cc.

#### 4.10.2.37 #define REQUEST__STOP(context) SPARK::Requests::TDispatcher::stop( REQUEST__HEADER( THIS, ACTIVE_PROBLEM, context ) )

Macro to send a stop request from an atomic class.

See SPARK::Requests::TDispatcher::stop()

**Parameters:**

***context*** is a `const char*` that contains the description of the calling context

#### 4.10.2.38 #define REQUEST__SET_STOP_TIME(context, time) SPARK::Requests::TDispatcher::set_stop_time( REQUEST__HEADER( THIS, ACTIVE_PROBLEM, context ), time )

Macro to send a set stop time request from an atomic class.

See SPARK::Requests::TDispatcher::set_stop_time()

**Parameters:**

***context*** is a `const char*` that contains the description of the calling context

***time*** is a double `value` that speicifies the desired stopping time for the simulator

#### 4.10.2.39 #define REQUEST__REPORT(context) SPARK::Requests::TDispatcher::report( REQUEST__HEADER( THIS, ACTIVE_PROBLEM, context ) )

Macro to send a report request from an atomic class.

See SPARK::Requests::TDispatcher::report()

**Parameters:**

***context*** is a `const char*` that contains the description of the calling context

#### 4.10.2.40 #define REQUEST__SNAPSHOT(context, filename) SPARK::Requests::TDispatcher::snapshot( REQUEST__HEADER( THIS, ACTIVE_PROBLEM, context ), filename )

Macro to send a snapshot request from an atomic class.

See SPARK::Requests::TDispatcher::snapshot()

**Parameters:**

***context*** is a `const char*` that contains the description of the calling context

***filename*** is a `const char*` that contains the file name used to generate the problem snapshot

#### 4.10.2.41 #define REQUEST__SET_MEETING_POINT(context, time) SPARK::Requests::TDispatcher::set_meeting_point( REQUEST__HEADER( THIS, ACTIVE_PROBLEM, context ), time )

Macro to send a request to synchronize with a meeting point from an atomic class.

See SPARK::Requests::TDispatcher::set_meeting_point()

**Parameters:**

***context*** is a `const char*` that contains the description of the calling context

***time*** is a `double` value that indicates the meeting point the simulator will try to synchronize with

#### 4.10.2.42 #define REQUEST__CLEAR_MEETING_POINTS(context) SPARK::Requests::TDispatcher::clear_meeting_points( REQUEST__HEADER( THIS, ACTIVE_PROBLEM, context ) )

Macro to send a clear meeting points request from an atomic class.

See SPARK::Requests::TDispatcher::clear_meeting_points()

**Parameters:**

***context*** is a `const char*` that contains the description of the calling context

#### 4.10.2.43 #define REQUEST__RESTART(context) SPARK::Requests::TDispatcher::restart( REQUEST__HEADER( THIS, ACTIVE_PROBLEM, context ) )

Macro to send a restart request from an atomic class.

See SPARK::Requests::TDispatcher::restart()

**Parameters:**

***context*** is a `const char*` that contains the description of the calling context

#### 4.10.2.44 #define REQUEST__SET_DYNAMIC_STEPPER(context, stepper) SPARK::Requests::TDispatcher::set_dynamic_stepper( REQUEST__HEADER( THIS, ACTIVE_PROBLEM, context ), stepper )

Macro to send a "set dynamic stepper" request from an atomic class.

See SPARK::Requests::TDispatcher::set_dynamic_stepper()

**Parameters:**

***context*** is a `const char*` that contains the description of the calling context.

***stepper*** is a `TDynamicStepper` object that defines the structure of the dynamic steps.

**Warning:**

Not implemented yet.

#### 4.10.2.45 #define REQUEST__SET_TIME_STEP(context, h) SPARK::Requests::TDispatcher::set_time_step( REQUEST__HEADER( THIS, ACTIVE_PROBLEM, context ), h )

Macro to send a "set time step" request from an atomic class.

See SPARK::Requests::TDispatcher::set_time_step()

**Parameters:**

***context*** is a `const char*` that contains the description of the calling context

***h*** is a `double` value that indicates the next candidate time step to try

**Examples:**

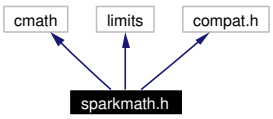
integrator_euler.cc.

## 4.11 sparkmath.h File Reference

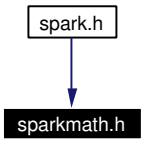
Math functions.

```
#include <cmath>
#include <limits>
#include "compat.h"
```

Include dependency graph for sparkmath.h:




This graph shows which files directly or indirectly include this file:




### Namespaces

- namespace SPARK

### 4.11.1 Detailed Description

Math functions.

Declares constant variables :

- SPARK::TINY
- SPARK::SQRT_UROUND

Declares functions :

- SPARK::min()
- SPARK::max()
- SPARK::sign()
- SPARK::abs()
- SPARK::InfiniteOrNaN()
- SPARK::log2()

**Author:**
Dimitri Curtil (LBNL/SRG)

**Date:**
May 21, 2002

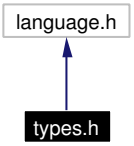
**Attention:**

## 4.12 types.h File Reference

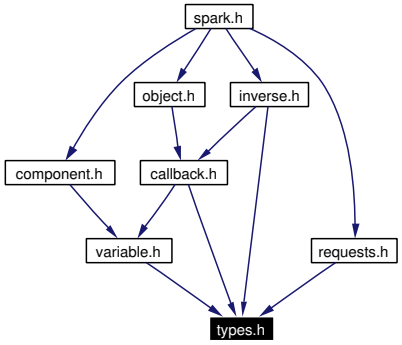
Declaration of the type enums describing the various entities in a SPARK problem.

```
#include "language.h"
```

Include dependency graph for types.h:




This graph shows which files directly or indirectly include this file:




### Namespaces

- namespace SPARK

### 4.12.1 Detailed Description

Declaration of the type enums describing the various entities in a SPARK problem.

Declaration of :

- atomic class types in SPARK::AtomicTypes
- function types in SPARK::FunTypes
- callback types in SPARK::CallbackTypes
- callback return types in SPARK::ReturnTypes
- callback prototypes in SPARK::ProtoTypes
- variable types in SPARK::VariableTypes
- request types in SPARK::RequestTypes

**Attention:**



**Author:**

Dimitri Curtil (LBNL/SRG)

**Date:**

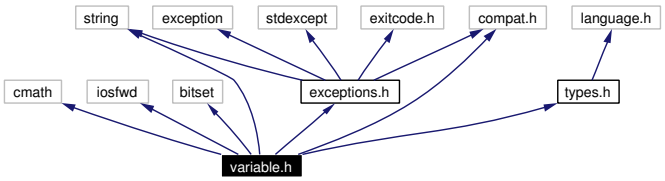
January 5, 2003

## 4.13 variable.h File Reference

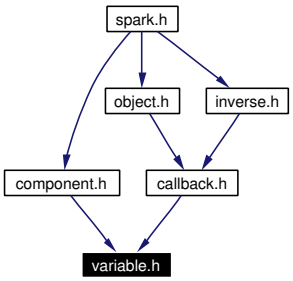
Declaration of the classes and types used to describe the problem variables.

```
#include <cmath>
#include <iosfwd>
#include <bitset>
#include <string>
#include "types.h"
#include "exceptions.h"
#include "compat.h"
```

Include dependency graph for variable.h:




This graph shows which files directly or indirectly include this file:




### Namespaces

- namespace SPARK
- namespace SPARK::Variable

### 4.13.1 Detailed Description

Declaration of the classes and types used to describe the problem variables.

Class declaration :

- SPARK::TVariable
- SPARK::TUnknown
- SPARK::Variable::TInterface

Function declaration :

- SPARK::Variable::ComputeScale()
- SPARK::Variable::Write()

**Author:**
Dimitri Curtil (LBNL/SRG)

**Date:**
May 15, 2002

**Attention:**

# Chapter 5

# SPARK Atomic Class API Example Documentation

## 5.1 analytical_frst_ord.cc

This is an example of how to use the various macro definitions to manage private data in an atomic class. In particular, we demonstrate the usage of:

- SET_DATA
- GET_DATA
- DELETE_DATA

This atomic class computes the analytical solution (a.k.a. closed-form solution) of the first-order, constant coefficient, linear, homogeneous ODE.

For more details, consult http://oregonstate.edu/dept/math/CalculusQuestStudyGuides/ode/first/linear/linear.html

```
/// analytical_frst_ord.cc
/// Atomic class that computes the analytical solution (a.k.a. closed-form solution)
/// of the first-order, constant coefficient, linear, homogeneous ODE.
///
/// ODE:
///     xdot = B - A*x
///
/// Initial conditions:
///     x(t_IC) = x_IC
///
/// Analytical solution: (can be used to compute the true integration error)
/// For more details, consult
/// http://oregonstate.edu/dept/math/CalculusQuestStudyGuides/ode/first/linear/linear.html
///
///     x(t) = C * exp(-A*(t-t_IC)) + B/A
/// where:
///   C + b/a = x_IC
///       b/a = lim x (t->inf)
///
/// where :
///    x    : dynamic variable
///    xdot : first derivative of x
///
///    A    : constant coefficient
///    B    : constant coefficient
```

```
///
//////////////////////////////////////////////////////////////////////////


#ifdef SPARK_PARSER

PORT time "time" [s];
PORT A    "A";
PORT B    "B";
PORT x    "x";
PORT xdot "xdot";
PORT x_IC "initial condition for x";

EQUATIONS {
        x, xdot = analytical_frst_ord( time, x_IC, A, B );
}

FUNCTIONS {
        x, xdot = analytical_frst_ord__evaluate( time )
                CONSTRUCT = analytical_frst_ord__construct( time, x_IC, A, B )
                DESTRUCT  = analytical_frst_ord__destruct( )
                ;
}

#endif  /*SPARK_PARSER*/


#include <strstream>
using std::ostrstream;
using std::ends;

#include <iostream>
using std::endl;

#include "spark.h"


// Class that calculates the analytical solution of a 1st-order
// ODE with constant, linear coefficients. The equations
// describing the analytical solution are constructed in the class
// constructor.
class TData {
public:
    // Structors
    TData(double t_IC, double x_IC, double a, double b)
        : T_IC(t_IC), C(x_IC - B/A), A(a), B(b)
    {}
    ~TData()
    {}

    // Main methods
    double x(double time) { return C * exp(-A*(time-T_IC)) + B/A; }
    double xdot(double time) { return -A * C * exp(-A*(time-T_IC)); }

    double GetA() const { return A; }
    double GetB() const { return B; }
    double GetC() const { return C; }
    double GetT_IC() const { return T_IC; }

    private:
    // Private data
    double T_IC;
    double A ;
    double B ;
    double C;
};


EVALUATE( analytical_frst_ord__evaluate )
{
```

```
    ARGUMENT( 0, time );
    TARGET( 0, x );
    TARGET( 1, xdot );

    // Retrieve private data
    GET_DATA( THIS, TData, MyData );
    //  The previous line is equivalent to:
    //  TData* MyData = static_cast<TData*>( THIS->GetData() );

    // Set target values
    x    = MyData->x( time );
    xdot = MyData->xdot( time );
}

CONSTRUCT( analytical_frst_ord__construct )
{
    ARGUMENT( 0, time );
    ARGUMENT( 1, x_IC );
    ARGUMENT( 2, A );
    ARGUMENT( 3, B );

    // Allocate instance private data
    TData* MyData = new TData( time, x_IC, A, B );

    // Check whether memory was allocated correctly or not
    if ( MyData == 0 ) {
        REQUEST__ABORT( "Could not allocate instance private data!" );
    }

    // Store pointer to private data within this object
    SET_DATA( THIS, TData, MyData );
    //  The previous line is equivalent to:
    //  THIS->SetData( MyData );

    // Report equation string for analytical solution to run log file
    ostrstream Text;
    Text << "x(t) = " << MyData->GetC() << " * exp(" << MyData->GetA()
         << "*( time - " << MyData->GetT_IC() << ") ) + "
         << MyData->GetB()/MyData->GetA() << ends;

    RUN_LOG( Text.str() );
}

DESTRUCT( analytical_frst_ord__destruct )
{
    // Release allocated memory
    // (the false flag indicates that this not a built-in array type)
    DELETE_DATA( THIS, TData, false );
    // The previous line is equivalent to:
    //  TData* MyData = static_cast<TData*>( THIS->GetData() );
    //  if ( MyData ) {
    //          delete MyData;
    //  }
}
```

## 5.2 analytical_spring.cc

This example shows how to use the ARGUMENT, TARGET, ERROR_LOG and THIS macro definitions to implement a multi-valued inverse function with private data in an atomic class.

In particular, this atomic class implements the following callbacks:

- CONSTRUCT
- DESTRUCT

Consult the following website for a detailed explanation of how the equations implemented in this class have been derived: http://oregonstate.edu/dept/math/CalculusQuestStudyGuides/ode/second/so_lin_-homocc/so_lin_homocc.html

```
/// analytical_spring.cc
/// Atomic class that computes the analytical solution (a.k.a. closed-form solution)
/// of the second-order, constant coefficient, linear, homogeneous ODE.
///
/// Second order differential equation modeling simple harmonic motion,
/// as might be described by a vibrating spring, whose motion is resisted by
/// a force proportional to velocity.
///
/// ODE:
///    ddy + c*dy + k*y = 0
///
/// where :
///    y   : displacement of a unit mass attached to the end of the spring
///    dy  : first derivative of y
///    ddy : second derivative of y
///
///    c   : damping constant
///    k   : stiffness constant
///
////////////////////////////////////////////////////////////////////////////////

#ifdef SPARK_PARSER

PORT time "time" [s];

PORT k "stiffness constant" [1/s^2];
PORT c "damping constant" [1/s];

// Note it would be possible to use any other distance unit in place of [m]
// as long as the same unit is consistently used for y, dy, and ddy
PORT y   "displacement" [m];
PORT dy  "velocity" [m/s];
PORT ddy "acceleration" [m/s^2];

PORT y_IC  "displacement (initial condition)" [m];
PORT dy_IC "velocity (initial condition)" [m/s];


EQUATIONS {
    y, dy, ddy = analytical_spring__evaluate( k, c );
}

FUNCTIONS {
    y, dy, ddy = analytical_spring__evaluate( time )
        CONSTRUCT = analytical_spring__construct( time, y_IC, dy_IC, k, c )
        DESTRUCT  = analytical_spring__destruct( )
        ;
}

#endif  /*SPARK_PARSER*/

// System includes
```

```
#include <iostream>
using std::endl;

#include <strstream>
using std::ostrstream;
using std::ends;

#include <string>
using std::string;


#include "spark.h"


////////////////////////////////////////////////////////////////////////////
namespace analytical_spring {

////////////////////////////////////////////////////////////////////////
/// Class that computes the closed-form solution of the linear, 2nd order ODE
/// assuming that the initial conditions passed to the constructor are
/// consistent and that the constants k and c remain unchanged throughout the
/// simulation. If the constants k and c change, then you should construct a
/// anew instance of the class TData with the new values passed to the constructor.
///
/// General form of ths solution when characterictic polynomial has 2 distinct
/// real roots ALPHA and BETA:
///    y(t) = C1*exp(ALPHA*t) + C2*exp(BETA*t)
///
/// Initial conditions:        ddy(t0) + c*dy(t0) + k*y(t0) = 0
/// Characteristic polynomial: c^2-4k > 0
///
/// Distinct real roots (both negative):
///    ALPHA = 0.5*( -c + sqrt(c^2-4k) )
///    BETA  = 0.5*( -c - sqrt(c^2-4k) )
///
class TData {
public:
    ////////////////////////////////////////////////////////////////////
    // Structors
    ////////////////////////////////////////////////////////////////////
    TData(double time0, double y0, double dy0, double k, double c);
    ~TData();

    ////////////////////////////////////////////////////////////////////
    // Access methods
    ////////////////////////////////////////////////////////////////////
    double GetALPHA() const { return ALPHA; }
    double GetBETA() const { return BETA; }

    double GetC1() const { return C1; }
    double GetC2() const { return C2; }

    double y(double time) const ;
    double dy(double time) const ;
    double ddy(double time) const ;

private:
    double ALPHA;
    double BETA;
    double C1;
    double C2;

};
////////////////////////////////////////////////////////////////////////////


}; // namespace analytical_spring
////////////////////////////////////////////////////////////////////////////
```

```
///////////////
// CALLBACKS //
///////////////


CONSTRUCT( analytical_spring__construct )
{
    ARGUMENT( 0, time );
    ARGUMENT( 1, y_IC );
    ARGUMENT( 2, dy_IC );
    ARGUMENT( 3, k );
    ARGUMENT( 4, c );

    analytical_spring::TData* MyData = 0;

    try {
        MyData = new analytical_spring::TData(
            time,
            y_IC.GetInit(),
            dy_IC.GetInit(),
            k,
            c
        );
    }

    // Process exception thrown by constructor if any
    catch (const string& Msg) {
        REQUEST__ABORT( Msg.c_str() );
    }


    // Abort simulation if memory could not be allocated!
    if ( !MyData ) {
        REQUEST__ABORT( "Could not allocate memory for my private data!" );
    }


    // Store pointer to private data within this object
    THIS->SetData( MyData );


    // Report equation string for analytical solution to run log file
    ostrstream Text;
    Text << "y(t) = (" << MyData->GetC1() << " * exp(" << MyData->GetALPHA()
         << "*t)) + (" << MyData->GetC2() << " * exp(" << MyData->GetBETA()
         << "*t))" << ends;

    RUN_LOG( Text.str() );
}


EVALUATE( analytical_spring__evaluate )
{
    ARGUMENT( 0, time );
    TARGET( 0, y );
    TARGET( 1, dy );
    TARGET( 2, ddy );

    // Cast void* to private data type
    analytical_spring::TData* MyData = static_cast<analytical_spring::TData*>(
        THIS->GetData()
    );

    // Set target values
    y   = MyData->y( time );
    dy  = MyData->dy( time );
    ddy = MyData->ddy( time );
}
```

```
DESTRUCT( analytical_spring__destruct )
{
    // Cast void* to private data type
    analytical_spring::TData* MyData = static_cast<analytical_spring::TData*>(
        THIS->GetData()
    );

    // Release allocated memory
    if ( MyData ) {
        delete MyData;
    }
}



//////////////////
// PRIVATE DATA //
//////////////////

using analytical_spring::TData;


// Function name        : TData::TData
// Description      :
// Return type          :
// Argument         : double time0
// Argument         : double y0
// Argument         : double dy0
// Argument         : double k
// Argument         : double c
TData::TData(double time0, double y0, double dy0, double k, double c)
{
    double Discriminant = c*c - 4.0*k;

    if ( Discriminant > 0.0 ) { // strongly damped spring
        //
        // y(t) = C1*exp(ALPHA*t) + C2*exp(BETA*t)
        //
        double Temp = sqrt(Discriminant);
        ALPHA = 0.5*(-c + Temp);
        BETA  = 0.5*(-c - Temp);

        // Initial conditions
        //
        // C1*exp(ALPHA*time0) + C2*exp(BETA*time0) = y(time0) = y0
        // C1*ALPHA*exp(ALPHA*time0) + C2*BETA*exp(BETA*time0) = dy(time0) = dy0
        //
        const double Determinant = BETA - ALPHA;
        const double A_11 = exp(ALPHA*time0);
        const double A_12 = exp(BETA*time0);
        const double A_21 = ALPHA*exp(ALPHA*time0);
        const double A_22 = BETA*exp(BETA*time0);

        C1 = 1.0/Determinant * ( A_22*y0 - A_12*dy0 );
        C2 = 1.0/Determinant * (-A_21*y0 + A_11*dy0 );
    }
    else {
        string ErrMsg("Could not construct analytical_spring::TData object: ");
        ErrMsg += "characteristic polynomial does not have 2 distinct real roots!";
        throw ErrMsg;
    }
}


// Function name        : TData::~TData
// Description      :
// Return type          :
TData::~TData()
{
```

```
    // Nothing to do
}


// Function name        : TData::y
// Description      :
// Return type          : double
// Argument         : double time
double TData::y(double time) const
{
    return (
        C1 * exp(ALPHA * time) +
        C2 * exp(BETA * time)
    );
}


// Function name        : TData::dy
// Description      :
// Return type          : double
// Argument         : double time
double TData::dy(double time) const
{
    return (
        C1 * ALPHA * exp(ALPHA * time) +
        C2 * BETA * exp(BETA * time)
    );
}


// Function name        : TData::ddy
// Description      :
// Return type          : double
// Argument         : double time
double TData::ddy(double time) const
{
    return (
        C1 * ALPHA * ALPHA * exp(ALPHA * time) +
        C2 * BETA * BETA * exp(BETA * time)
    );
}
```

## 5.3 integrator_euler.cc

This is an example of how to use the various macro definitions to send requests from an atomic class.

```
// integrator_euler.cc
//
// Atomic class that defines Euler integrator defined as CLASSTYPE INTEGRATOR.
//
// - The initial condition for the dynamic variable x is returned for a static step at initial time
// - The integration scheme is invoked for a dynamic step
// - If VariableTimeStep ( 1 ()) in the *.run file, then the time step is adapted in order
//   to satisfy the integration tolerance specified in the GlobalSettings section of the
//   *.prf file with the key Tolerance.
//
////////////////////////////////////////////////////////////////////////


#ifdef  spark_parser

CLASSTYPE  INTEGRATOR ;

PORT    x;
PORT    xdot;
PORT    dt;

EQUATIONS {
    x = integrator_euler( xdot );
    bad_inverses = xdot, dt ;
}

FUNCTIONS {
    x = EVALUATE                    integrator_euler__evaluate( xdot, dt )
        PREDICT                     integrator_euler__predict( xdot, dt )

        STATIC_CONSTRUCT            integrator_euler__static_construct()
        CONSTRUCT                   integrator_euler__construct( x, xdot )

        STATIC_PREPARE_STEP         integrator_euler__static_prepare_step()
        PREPARE_STEP                integrator_euler__prepare_step( x, xdot, dt )

        CHECK_INTEGRATION_STEP      integrator_euler__check_integration_step( x )
        STATIC_CHECK_INTEGRATION_STEP integrator_euler__static_check_integration_step()

        COMMIT                      integrator_euler__commit( x, xdot )
        STATIC_COMMIT               integrator_euler__static_commit()

        ROLLBACK                    integrator_euler__rollback( x, xdot )
        STATIC_ROLLBACK             integrator_euler__static_rollback()

        DESTRUCT                    integrator_euler__destruct()
        STATIC_DESTRUCT             integrator_euler__static_destruct()
    ;
}


#endif  // spark_parser


// The debugger can’t handle symbols more than 255 characters long.
// STL often creates symbols longer than that.
// When symbols are longer than 255 characters, the warning is disabled.
#pragma warning(disable:4786)

#include "spark.h"


//////////////////
// PRIVATE DATA //
//////////////////
```

```
////////////////////////////////////////////////////////////////////////////
// Include implementation files with support for the integration methods
////////////////////////////////////////////////////////////////////////////
#include "integrators/euler.h"


////////////////////////////////////////////////////////////////////////////
// Activate the namespace of the desired integration method
////////////////////////////////////////////////////////////////////////////
using namespace SPARK::Euler;


////////////////////////////////////////////////////////////////////////////
// Define shortcut types
////////////////////////////////////////////////////////////////////////////
typedef TIntegrationMethod::integration_controller_type  TIntegrationController;
typedef TIntegrationMethod::integrator_type              TIntegrator;



///////////////
// CALLBACKS //
///////////////


////////////////////////////////////////////////////////////////////////////
STATIC_CONSTRUCT( integrator_euler__static_construct )
{
    TIntegrationController* MyController = TIntegrationMethod::make_integration_controller(
        THIS->GetNumObjects(),     // Number of instances
        ACTIVE_PROBLEM->GetName(), // Problem name
        false                      // Generate verbose integration diagnostic?
    );

    // Abort simulation if memory could not be allocated!
    if ( !MyController ) {
        REQUEST__ABORT( "Could not allocate memory for the integration controller!" );
    }

    // Store pointer to private static data within this inverse
    SET_DATA( THIS, TIntegrationController, MyController );

}
////////////////////////////////////////////////////////////////////////////


////////////////////////////////////////////////////////////////////////////
CONSTRUCT( integrator_euler__construct )
{
    ARGUMENT( 0, x );
    ARGUMENT( 1, xdot );

    GET_DATA( ACTIVE_INVERSE, TIntegrationController, MyController );

    TIntegrator* MyIntegrator = TIntegrationMethod::make_integrator(
        THIS->GetInstanceHandle(), // Unique instance handle
        x,                         // Dynamic variable's name
        xdot,                      // Time-derivative's name
        MyController               // Integration controller
    );


    // Abort simulation if memory could not be allocated!
    if ( !MyIntegrator ) {
        REQUEST__ABORT( "Could not allocate memory for the integrator!" );
    }

    // Store pointer to private data within this object
```

```
    SET_DATA( THIS, TIntegrator, MyIntegrator );

}
////////////////////////////////////////////////////////////////////////////


////////////////////////////////////////////////////////////////////////////
STATIC_PREPARE_STEP( integrator_euler__static_prepare_step )
{
    // Retrieve integration step if dynamic step
    if ( !ACTIVE_PROBLEM->IsStaticStep() ) {

        GET_DATA( THIS, TIntegrationController, MyController);

        MyController->PrepareIntegrationStep( ACTIVE_PROBLEM->GetGlobalTimeStep() );
    }
}
////////////////////////////////////////////////////////////////////////////


////////////////////////////////////////////////////////////////////////////
PREPARE_STEP( integrator_euler__prepare_step )
{
    // Prepare integrator private data at beginning of integration step
    if ( !ACTIVE_PROBLEM->IsStaticStep() ) {

        ARGUMENT( 0, x );
        ARGUMENT( 1, xdot );
        ARGUMENT( 2, dt );

        GET_DATA( THIS, TIntegrator, MyIntegrator);

        MyIntegrator->PrepareIntegrationStep(
            x,
            xdot,
            dt
        );
    }
}
////////////////////////////////////////////////////////////////////////////


////////////////////////////////////////////////////////////////////////////
PREDICT( integrator_euler__predict )
{
    TARGET( 0, x );

    ARGUMENT( 0, xdot );
    ARGUMENT( 1, dt );

    double  result;

    if ( ACTIVE_PROBLEM->IsStaticStep() ) {  // Static calculation

        result = (
            ACTIVE_PROBLEM->IsInitialTime() ?
            x.GetInit() // Initial time solution special case
            :
            x[1] // Past value for restart after initial time solution
        );

    }
    else { // Compute prediction using past value of time-derivative

        GET_DATA( THIS, TIntegrator, MyIntegrator);

        result = MyIntegrator->Predict(
            x,
            xdot,
            dt
```

```
        );

    }

    // Update dynamic variable with result
    x = result;
}
////////////////////////////////////////////////////////////////////////////


////////////////////////////////////////////////////////////////////////////
EVALUATE( integrator_euler__evaluate )
{
    TARGET( 0, x );

    ARGUMENT( 0, xdot);
    ARGUMENT( 1, dt);

    double  result;

    if ( ACTIVE_PROBLEM->IsStaticStep() ) {  // Static calculation

        result = (
            ACTIVE_PROBLEM->IsInitialTime() ?
            x.GetInit() // Initial time solution
            :
            x[1] // Past value for warm restart
        );

    }
    else { // Perform the actual integration using current value of time-derivative

        GET_DATA( THIS, TIntegrator, MyIntegrator);

        result = MyIntegrator->Integrate(
            x,
            xdot,
            dt
        );

    }

    // Update dynamic variable with result
    x = result;
}
////////////////////////////////////////////////////////////////////////////


////////////////////////////////////////////////////////////////////////////
CHECK_INTEGRATION_STEP( integrator_euler__check_integration_step )
{
    // Estimate integration error only if:
    //  - variable time step mode
    //  - this is not a static step
    if ( ACTIVE_PROBLEM->IsStaticStep() ) {

        return true;

    }
    else {

        ARGUMENT( 0, x);

        GET_DATA( THIS, TIntegrator, MyIntegrator);

        MyIntegrator->EstimateLocalError( x );

        // Nothing to do. Decision is postponed to static check integration step callback
        return true;
    }
```

```
}
////////////////////////////////////////////////////////////////////////////


////////////////////////////////////////////////////////////////////////////
STATIC_CHECK_INTEGRATION_STEP( integrator_euler__static_check_integration_step )
{
    // Check integration error only if:
    //  - variable time step mode
    //  - this is not a static step
    if ( ACTIVE_PROBLEM->IsStaticStep() ) {

        return true;

    }
    else {

        if ( ACTIVE_PROBLEM->IsTimeStepVariable() ) {

            GET_DATA( THIS, TIntegrationController, MyController);

            return MyController->CheckIntegrationStep( THIS->GetTolerance()     );

        }
        else {

            return true;

        }
    }
}
////////////////////////////////////////////////////////////////////////////


////////////////////////////////////////////////////////////////////////////
COMMIT( integrator_euler__commit )
{
    GET_DATA( THIS, TIntegrator, MyIntegrator);

    if ( ACTIVE_PROBLEM->IsStaticStep() ) {

        MyIntegrator->Restart();

    }
    else {

        ARGUMENT( 0, x);
        ARGUMENT( 1, xdot);

        MyIntegrator->Commit(
            x,
            xdot
        );
    }
}
////////////////////////////////////////////////////////////////////////////


////////////////////////////////////////////////////////////////////////////
STATIC_COMMIT( integrator_euler__static_commit )
{
    // Predict next time step only if:
    //  - variable time step mode
    //  - this is not a static step
    GET_DATA( THIS, TIntegrationController, MyController);

    if ( ACTIVE_PROBLEM->IsStaticStep() ) {

        MyController->Restart();
```

```
    }
    else {

        if ( ACTIVE_PROBLEM->IsTimeStepVariable() ) {

            // Requests time step for next step to satisfy the user-requested
            // tolerance for the next dynamic step.
            REQUEST__SET_TIME_STEP(
                MyController->GetName(),
                MyController->PredictTimeStep( THIS->GetTolerance() )
            );

        }

        MyController->Commit();
    }
}
////////////////////////////////////////////////////////////////////////////


////////////////////////////////////////////////////////////////////////////
ROLLBACK( integrator_euler__rollback )
{
    GET_DATA( THIS, TIntegrator, MyIntegrator);

    if ( ACTIVE_PROBLEM->IsStaticStep() ) {

        MyIntegrator->Restart();

    }
    else {

        ARGUMENT( 0, x);
        ARGUMENT( 1, xdot);

        MyIntegrator->Rollback(
            x,
            xdot
        );
    }
}
////////////////////////////////////////////////////////////////////////////


////////////////////////////////////////////////////////////////////////////
STATIC_ROLLBACK( integrator_euler__static_rollback )
{
    // Correct current time step only if:
    //  - variable time step mode
    //  - this is not a static step
    GET_DATA( THIS, TIntegrationController, MyController);

    if ( ACTIVE_PROBLEM->IsStaticStep() ) {

        MyController->Restart();

    }
    else{

        if ( ACTIVE_PROBLEM->IsTimeStepVariable() ) {

            // Requests time step for current step after rejection to satisfy the user-requested
            // tolerance for the next dynamic step.
            REQUEST__SET_TIME_STEP(
                MyController->GetName(),
                MyController->CorrectTimeStep( THIS->GetTolerance() )
            );

        }
```

```
        MyController->Rollback();
    }
}
////////////////////////////////////////////////////////////////////////////////


////////////////////////////////////////////////////////////////////////////////
DESTRUCT( integrator_euler__destruct )
{
    // Deallocate instance private data
    DELETE_DATA(
        THIS,
        TIntegrator,
        false
    );
}
////////////////////////////////////////////////////////////////////////////////


////////////////////////////////////////////////////////////////////////////////
STATIC_DESTRUCT( integrator_euler__static_destruct )
{
    // Deallocate static private data
    DELETE_DATA(
        THIS,
        TIntegrationController,
        false
    );
}
////////////////////////////////////////////////////////////////////////////////
```

## 5.4 sum.cc

This is an example of how to use the various ARGDEF and RETURN macro definitions to implement a single-valued inverse function in an atomic class.

```
/*+++
  Identification:  SPARK sum object

  Abstract:
       Sums two links

  Acceptable input set:
       a = 2, b = 1
---*/
#ifdef SPARK_PARSER

PORT a  "Summand 1" ;
PORT b  "Summand 2" ;
PORT c  "Sum" ;

EQUATIONS {
        c = a + b ;
}

FUNCTIONS {
        a = sum__a_or_b( c, b ) ;
        b = sum__a_or_b( c, a ) ;
        c = sum__c( a, b ) ;
}

#endif /* SPARK_PARSER */
#include "spark.h"

////////////////////////////////////////////////////////
EVALUATE( sum__a_or_b )
{
    ARGDEF(0, c);
    ARGDEF(1, b);
    double a_or_b ;

    a_or_b = c - b;
    RETURN( a_or_b ) ;
}
////////////////////////////////////////////////////////
////////////////////////////////////////////////////////
EVALUATE( sum__c )
{
    ARGDEF(0, a);
    ARGDEF(1, b);
    double c;

    c = a + b;
    RETURN( c ) ;
}
////////////////////////////////////////////////////////
```

# Index