Institutt for energiteknikk
OECD Halden Reactor Project

The Software Bus Application Framework

REFERENCE MANUAL

SOFTWARE BUS DOCUMENTATION

Revision: 4.6

Institutt for energiteknikk
OECD Halden Reactor Project

The Software Bus development team can be contacted at the following e-mail address:
softbus@hrp.no
For further information, see the Software Bus WWW pages at URL:
http://www.ife.no/swbus

The information in this document is subject to change without notice and should not be construed as a commitment by Institutt for Energiteknkikk.

Institutt for Energiteknikk, OECD Halden Reactor Project, assumes no responsibility for any errors that may appear in this document.

Trademarks used within are the properties of the respective trademark owners.

Published by : Institutt for Energiteknikk, OECD Halden Reactor Project

Date : June 07

Revision : R4.6

Table Of Contents

Global Data Types

Description of global data types used in *SAF*.

Data Types

SafTBool	Boolean data type used in *SAF*.

Data Types

SafTBool

Description

SafTBool is the boolean data type used in *SAF*. It can have the values *SafTFalse* or *SafTTrue*.

SafCategory

SafCategory is a struct used in the external interface **SafReportExternalInterface**. It contains information about a category in the remote task.

The information stored in a **SafCategory** can be used to enable or disable information printed to an output device in a remote application.

The following categories are predefined in *SAF*:

- **SAF_CAT_ERROR**
- **SAF_CAT_WARNING**
- **SAF_CAT_DEBUG**
- **SAF_CAT_INFO**
- **SAF_CAT_FLOW**

For more information about how to create new categories refer to the **SafReport** macro section.

Refer to the **SafReportExternalInterface** description for more information about why and how this struct can be used.

Struct Methods

Attributes	
mFormat	Format string for this category.
mGroupId	Group identifier for this category.
mName	Human readable name for this category.
mString	Unique category identification.

Data Members

SafCategory::mFormat

Description

This is the format string used when printing a message of this category to an output device. The syntax for this attribute is described in the **SafReport** macro section. Refer to **SafDAddCategory**.

SafCategory::mGroupId

Description

All categories belong to a group. It can be a user defined group or one of the predefined groups declared in *SAF*. For more information about these group identifiers refer to the **SafReport** macro section.

SafCategory::mName

Description

This attribute is a description or a human readable name for the category. This name does not need to be unique. For more information about this attribute refer to the **SafReport** macro section.

SafCategory::mString

Description

This attribute is the name of the category used when printing a message. It must be a unique name. This is also the name of the category variable in the remote task. For more information about this attribute refer to the **SafReport** macro section.

SafConnection

The *SAF* class **SafConnection** is used as a base class when creating connection objects. A **SafConnection** object is used when establishing connection between two *SAF* compliant applications.

This class must be further sub-classed to enable overloading of methods. It is an abstract class. The following methods must be implemented in the **SafConnection** sub-class:

- **createInterface()**
- **destroyInterface()**

To establish a connection to a remote process the *SAF* client process must create a **SafConnection** sub-class object. The argument to the **SafConnection** constructor is the task name of the remote *SAF* task. After creating a **SafConnection** instance calling the member method **initiateConnection()** will start the process of establishing a connection to the remote process. Remember that connection between two *SAF* compliant tasks are not established before the virtual method **onConnect()** is invoked. This virtual method is invoked in both the client and the server module.

Class Methods

Construction/Destruction	
SafConnection	Constructs **SafConnection** object.
~SafConnection	Destroys **SafConnection** object.

Overridable Methods	
destroy	This method is called after the connection to the remote process has been terminated.
establish	This method is called before the connection to the remote process is established.
onConnect	Called when the connection to the remote process is established.
onConnectionBreak	Called when the connection to the remote process is broken.
onConnectionTimeout	Called when the task is unable to connect to the remote process.
onDisconnect	Called when the connection to the remote process is gracefully disconnected.
onIncompatibleVersions	Called when the process to establish connection to have an incompatible *SWBus* version.
onVariableUpdate	Called when receiving variable updates from external process. This method is not called from the framework. Must be supplied by the user.

Pure Overridable Methods

createInterface	Must create the internal interface based on the supplied interface class.
destroyInterface	Must delete the internal interface corresponding to the supplied interface identification.

Get Methods

connectionIsInitiated	Returns true if the connection to the remote process has been initiated.
isConnected	Returns true if connection is established.
isEstablished	Returns true if the instance's establish method has been called.
isNonBlocking	Returns true if the connection channel is set to non blocking mode.
remoteSTI	Returns the *SWBus STI* of the remote process.
taskName	Returns the name of the remote process.
taskClass	Returns the task class of the remote process.

Connection Methods

disconnect	Terminates the connection to the remote process.
enableNonBlocking	Enables non-blocking mode and sets the FIFO size.
initiateConnection	Initiates a connection to a remote process.

Interface Methods

externalInterfaceStore	Returns a reference to a list of external interfaces used on this channel.
findInterface	Returns an interface based on the interface identification.
internalInterfaceStore	Returns a reference to a list of internal interfaces used on this channel.
disconnectInterfaces	Calls the method **onDisconnect()** for all the interface objects in the list passed as parameter.

SWBus Methods

call	Calls an *SWBus* function in the remote process.
declare	Declares an *SWBus* function.
sub	Subclasses an *SWBus* class.

Macros

SafDConnectionInfo	Declaration of struct used as input parameter to **call()**.
SafDDestroyConnection	Calls the **destroy()** method.
SafDDeclareConnection	Declares functionality used by the methods **establish()** and **destroy()**.
SafDImplementConnection	Implements functionality used by the methods **establish()** and **destroy()**.

Global Functions

SafFGetConnection	Returns a pointer to a connection object.

Member Functions

SafConnection::call

```
SbTSTI call( SbTSTI function,
             SbTSTI inParam,
             SbTSTI outParam )
```

Description

This function is used to call a **SafConnection** sub-class method in the remote process. It can not call a **SafConnection** sub-class method directly, but indirectly through an *SWBus* function. Enough information is stored in the input parameter *inParam* to access the correct connection object in the remote process. This extra information is added to the input parameter in this method.

The parameter *inParam* must be either *SbCNull* or must have been created using the method **sub()**. If the parameter *inParam* is *SbCNull* an internal *SWBus* parameter is created storing information about the connection object to call in the remote process. In the remote process using the global function **SafFGetConnection()** will get a pointer to the correct connection object. The input parameter passed to this global function is the same input argument that was passed to the *SWBus* function.

If the *inParam* is not *SbCNull* the *SWBus* class and the corresponding *C* struct must have the same layout. It is extremely important that the first field in the struct is the macro **SafDConnectionInfo** or that it is derived directly from **SafConnectionClassStruct**, otherwise unpredictable behaviour may be the result.

The *outParam* can be a predefined *SWBus* class or a class created using the *SWBus* function **SbSub()**.

When calling an interface method in the remote process, always use the methods **declare()**, **sub()** and **call()**. Do not mix these functions with direct calls to **SbDeclare()**, **SbSub()** or **SbCall()**. It is extremely important to be aware of this because the function **SafFGetConnection()** can not be used to get correct connection object in the remote process if these functions are mixed. The exception is the output parameter. It is not necessary to use the method **sub()** to create the *SWBus* class because no extra information is used when returning from the *SWBus* function.

Note that even if a single variable of a predefined *SWBus* class should be passed as parameter to a remote function, it must be encapsulated in a user defined class created using the method **sub()**.

Access

This function is protected. Can only be called in sub-classes of **SafConnection**.

Parameters

function	*SWBus STI* of function to call in the remote process.
inParam	Input parameter to the function to call in the remote process.
outParam	Output parameter from the function in the remote process.

Return Value

SbCError is returned when one of the following situations occur:

- If the parameter *function* is not a valid *SbTSTI*.

- If the class for user defined functions can not be found.
- If location or creation of the input parameter object fails.
- If *inParam* does not identify an object of expected type.
- If the object identified by *in*Param does not contain any data.

In all other situations the value from **SbCall()** is returned.

Example

This example demonstrates how to call a function in a remote process. The function *remoteFunc* is the name of the *SWBus* function. In this example the class *cliConnection* is a sub-class of **SafConnection**.

```
void cliConnection::callMyFunc()
{
   / * Function takes no arguments */
   SbTSTI sti     = remoteSTI();
   SbTSTI rFuncId = SbId( sti, "remoteFunc"   );

   call( rFuncId, SbCNull, SbCNull );
}
```

The function *remoteCBFunc* is located in the remote process. The class *srvConnection* is a sub-class of **SafConnection**. Notice the use of **SafFGetConnection()** to get a pointer to the connection object.

```
SbTSTI remoteCBFunc( SbTSTI, SbTSTI, SbTSTI in, SbTSTI )
{
   srvConnection* c;
   c = (srvConnection *)SafFGetConnection( in );
   if ( c )
     c->myFunc();
   return SbCNull;
}
```

See Also: **SafConnectionClassStruct, SafFGetConnection, SafConnection::declare, SafConnection::sub**

SafConnection::connectionIsInitiated

`SafTBool connectionIsInitiated() const`

Description

A connection is defined as initiated when its **initiateConnection()** function has been called.

Access

This function is public.

Return Value

Returns *SafTTrue* if **initiateConnection()** has been called, otherwise *SafTFalse*.

Example

The following example demonstrates the use of **connectionIsInitiated()** to prevent that **initiateConnection()** is called more than once.

```
SafStatus cliConnection::connect()
{
  …
   if ( connectionIsInitiated() )
      return Saf OK;
   return initiateConnection();
}
```

See Also: SafConnection::initaiteConnection

SafConnection::createInferface

```
virtual SafStatus createInterface( const char* interfaceClass,
                                   int32       interfaceId ) =0
```

Description

This virtual method is a callback function that is automatically invoked when initialising an external interface in an external task. The *interfaceClass* passed as argument to this method is the same interface class that was passed as argument to the external interface when creating it in the remote task.

Note that the interface class should be hard coded in the **SafExternalInterface** sub-class, because this interface class is used by **createInterface()** to distinguish between different internal interfaces to create. **createInterface()** can only create internal interfaces it knows about at design time.

To respond to this request this method should always create an internal interface corresponding to the external interface in the remote process. The *interfaceId* is an identifier generated by the framework and must not be changed in any circumstance. This identifier must be supplied to the **SafInternalInterface** base class.

This function is pure virtual and must always be implemented in the **SafConnection** sub-class.

If the task is a server for other clients, this method must create the internal interface object and call the **SafInternalInterface** method **initialize()**. If the task is just a client, this method may just return *Saf_OK*.

The *interfaceClass* passed as argument to this function is used to distinguish between the different internal interfaces the **SafConnection** sub-class object can create. This string must uniquely identify an interface class.

Notice that the internal interface method **initialize()** must be called before returning from this function.

Access

This is a pure protected virtual public method. Must be implemented in sub-classes of **SafConnection**. Do not try to call this method directly from the application source code.

Parameters

interfaceClass	Unique name of the interface class known by the external interface and by the **SafConnection** sub-class method **createInterface**().
interfaceId	*SAF* generated identifier. Must be supplied to the **SafInternalInterface** base class.

Return Value

Should return *Saf_OK* if an internal interface object is created and properly initialised, otherwise return an error code.

Example

This example demonstrates an implementation of **createInterface()** that supports two interfaces, *CONSUMER* and *MONITOR*. The classes *consumerInterface* and *monitorInterface* inherit from **SafInternalInterface**.

```
SafStatus Connection::createInterface( const char* interfaceClass,
                                       int32       interfaceId )
{
   SafInternalInterface* iFC;

   if ( strcmp( interfaceClass, "CONSUMER" ) == 0 )
      iFC = new consumerInterface( this, interfaceClass,
                                         interfaceId );
   else
   if ( strcmp( interfaceClass, "MONITOR" ) == 0 )
      iFC = new monitorInterface( this, interfaceClass,
                                        interfaceId );
   else
      return InterfaceError; // Unknown interface. Could not create.

   iFC->initialize();        // Initialise the interface

   return Saf OK;
}
```

See Also: SafInternalInterface, SafConnection::destroyInterface

SafConnection::declare

```
static SbTSTI declare( const char* name,
                       SbTSTI  inputClass,
                       SbTSTI  outputclass,
                       SbTCode function )
```

Description

This method declares an *SWBus* function class called *name*. Instances of this class are callable functions that can be used as arguments to the method **call()**. *inputClass* and *outputClass* specify the classes of the formal input and output parameters respectively. The actual function code is situated at the address *function*.

If the parameter *inputClass* is set to *SbCNull* an internal *SWBus* class is declared as input to the *C* function. In that case the input parameter to the method **call()** must also be called with *SbCNull*.

declare() should be used when declaring an *SWBus* function. Do not mix the methods **declare()**, **sub()** and **call()** with a direct call to **SbDeclare()**, **SbSub()** or **SbCall()**. It is extremely important to be aware of this because the function **SafFGetConnection()** can not be used to get the correct connection object in the remote process if these functions are mixed. The exception is the output parameter. It is not necessary to use the method **sub()** to create the *SWBus* class because no extra information is used when returning from the *SWBus* function.

Access

This function is a static public method. It can be accessed without having an instance of a **SafConnection** sub-class object.

Parameters

name	Name of the function class
inputClass	Class of formal input parameter
outputclass	Class of formal output parameter
function	Function code address

Return Value

Returns *SbCError* if the function fails to get the *STI* of the predefined function class, otherwise the return value from **SbDeclare()**.

Example

This example declares a function taking the class *mInputArgStruct* as input parameter and a string as output parameter.

```
void Connection::establish()
{
   inputArgClass = sub( "mInputArgStruct" );

    … /* Skipping code for adding fields */

   decFunc = declare( "C Func", inputArgClass, SbCCString, C Func );
}


SbTSTI C Func( SbTSTI, SbTSTI, SbTSTI in, SbTSTI out )
{
   … /* Skipping code */

   return out;
}
```

See Also: SafConnection::call, SafConnection::sub, SbDeclare, SbId

SafConnection::destroy

`virtual SafStatus destroy()`

Description

This virtual method is called when a connection object is deleted and the macro **SafDDestroyConnection()** is put into the destructor of the **SafConnection** sub-class.

Normally *SWBus* classes and functions created in the method **establish()** should be deleted in this method.

It is possible to have this method called only when deleting the last connection object of a specific sub-class of **SafConnection**. To get this functionality use the macros **SafDDeclareConnection()** and **SafDImplementConnection()**. For more information about these macros refer to the macro section. When using these macros the method **establish()** will only be called when creating the first instance of a sub-class of **SafConnection**.

Access

This is a virtual protected method.

Return Value

Should return *Saf_OK* on success, otherwise an error code.

Example

This example demonstrates the use of the macros **SafDDeclareConnection**, **SafDImplementConnection** and **SafDDestroyConnection**. In the method **establish()**

SWBus objects are created. This method is called only once for the class *ConClass*. When the last instance of the class *ConClass* is deleted the method **destroy()** will be called. This method should destroy *SWBus* objects created in the method **establish()**.

```
class ConClass : public SafConnection
{
   …
   ~ConClass();
   SafStatus establish();
   SafStatus destroy();

   SafDDeclareConnection;
};


SafDImplementConnection( ConClass );


ConClass::~ConClass()
{
   …
   SafDDestroyConnection;
}

SafStatus ConClass::establish()
{
   /* Create SWBus classes and functions */

   return Saf OK;
}

SafStatus ConClass::destroy()
{
   /* Destroy SWBus classes and functions created in establish */

   return Saf OK;
}
```

See Also: **SafConnection::establish, SafDDeclareConnection, SafDImplementConnection, SafDDestroyConnection**

SafConnection::destroyInterface

`virtual SafStatus destroyInterface( int32 interfaceId ) =0`

Description

This pure virtual function must be implemented in the **SafConnection** sub-class. Normally this method should remove the internal interface object associated with the supplied *interfaceId*.

Access

This is a pure protected virtual method. Must be implemented in sub-classes of **SafConnection**. Do not try to call this method directly from the application source code.

Parameters

interfaceId The *SAF* generated identification of the interface to destroy.

Return Value

This function must return *Saf_OK* if the interface is successfully deleted, otherwise an error code.

Example

This example demonstrates the use of **internalInterfaceStore()** and the **remove**() method to destroy an internal interface.

```
SafStatus MonitorConn::destroyInterface( int32 iId )
{
   SafInterface* ii = internalInterfaceStore().contains( iId );
   delete ii;

   return Saf OK;
}
```

See Also: SafConnection::internalInterfaceStore

SafConnection::disconnect

`virtual SafStatus disconnect()`

Description

This method closes the connection to the remote process. **disconnect()** is automatically invoked by *SAF* within the **SafConnection** destructor function.

This function disconnects from the remote process and deletes the *SWBus* object identifying this connection.

Access

This function is public.

Return Value

Saf_SbNotConnected if currently not connected, otherwise *Saf_OK*.

See Also: SafConnection::initiateConnection

SafConnection::disconnectInterfaces

`SafStatus disconnectInterfaces( SafInterfaceStore&` *`interfaceListP`* **`)`**

Description

This method calls the **onDisconnect()** method for all the interface objects in the list *interfaceListP*.

Access

This function is protected. Can only be called from sub-classes of **SafConnection**.

Parameters

interfaceListP A list of internal or external interface objects.

Return Value

Returns *Saf_OK* if all the **SafInterface onDisconnect()** methods returned *Saf_OK*, otherwise the error code for the last interface that failed**.**

See Also: SafInterface::onDisconnect, SafInterfaceStore

SafConnection::enableNonBlocking

SafStatus enableNonBlocking(uint32 *fifoSize* **)**

Description

This method enables non-blocking mode on the connection. This function must be called after the connection to the remote task has been established. If the connection is broken and later restored this method must be called again to enable non-blocking.

It is the *SWBus* function **SbNonBlocking()** that is used to enable non-blocking.

Access

This function is public.

Parameters

fifoSize Size of TCP/IP FIFO buffer in bytes.

Return Value

Returns *Saf_SbNotConnected* if not connected, otherwise *Saf_OK*.

Example

In this example the method **enableNonBlocking()** is called when the connection to the remote task is established.

```
SafStatus myConnection::onConnect()
{
   enableNonBlocking( 65535 );
   …
   return Saf OK;
}
```

See Also: SafInterface::onDisconnect, SafInterfaceStore

SafConnection::establish

virtual SafStatus establish()

Description

This function is automatically invoked by *SAF* when a process initiates a connection by calling **initiateConnection()**. It may be overloaded in **SafConnection** sub-classes. It is always invoked before the virtual method **onConnect()**. By default **establish()** will be called each time **initateConnection()** is called for instances of a specific **SafConnection** sub-class.

To prevent that the method **establish()** is invoked each time **initiateConnection()** is called for a new interface instance the following macros can be used: **SafDDeclareConnection** and **SafDImplementConnection**. If these macros are used the **establish()** method will be invoked only once for each specific sub-class of **SafConnection**.

Normally this method should declare *SWBus* classes and functions relevant for the **SafConnection** sub-class.

Note that on the time when this method is invoked no connection is established to the remote task. Do not try to get any *SWBus STI*s to objects in the remote process in this method. A connection is not established before the **onConnect()** method is invoked. It is guaranteed that the **establish()** method is invoked in the server and client before the **onConnect()** method is called.

Access

This is a virtual protected method.

Return Value

The default implementation of the **establish()** method returns *Saf_OK*. The overloaded method should return *Saf_OK* on success, otherwise an error code.

Example

This example uses the **establish()** method to create the *SWBus* class *mMStringInput*.

```
SafStatus myConnection::establish()
{
   // Create SWBus classes and functions needed by this class.
   if ( !SBCSetStrIn )
   {

     SBCSetStrIn = sub( "mMStringInput" );
     if ( SbDBad( SBCSetStrIn ) )
     {
       cout << "Error: Failed to create class" << endl;
       return Saf_Error;
     }

     SbTSTI SBCField = SbAdd( SBCSetStrIn, SbCCString,
                              "string", 0 );
     if ( SbDBad( SBCField ) )
     {
        cout << "Error: Failed to add field" << endl;
        return Saf Error;
     }
   }

   return Saf OK;
}
```

See Also: **initiateConnection(), SafDDeclareConnection, SafDImplementConnection**

SafConnection::externalInterfaceStore

`SafInterfaceStore& externalInterfaceStore()`

Description

This method gets a reference to the list of external interfaces.

Access

This function is public.

Return Value

Returns a reference to the list of external interfaces associated with the connection.

Example

This example demonstrates the use of **externalInterfaceStore()** and the member method **remove()** to destroy an external interface.

```
void myConnection::remove( int32 iId )
{
   externalInterfaceStore().remove( iId, SafTTrue );
}
```

See Also: **SafInterfaceStore**

SafConnection::findInterface

```
SafInterface* findInterface( int32 interfaceId,
                             int32 interfaceType )
```

Description

This method searches for an interface element with the supplied *interfaceId*. The parameter *interfaceType* resolve whether to use the internal or the external interface list. If the supplied *interfaceType* is *SafCInternal,* the list of internal interfaces is traversed. If *interfaceType* is *SafCExternal*, the list of external interfaces is traversed

Access

This function is public.

Parameters

interfaceId	Interface identification of element to search for.
interfaceType	Interface type to search for. Must be set to either *SafCInternal* or *SafCExternal*.

Return Value

If the interface with the given *interfaceId* and *interfaceType* is found a pointer to the object is returned, otherwise NULL.

Example

This example demonstrates the use of **findInterface()** to search for an internal interface. The class *IInterface* is a sub-class of **SafInternalInterface**.

```
IInterface* Connection::find( int32 interfaceId )
{
   SafInterface* interfacePtr;
   interfacePtr = findInterface( interfaceId, SafCInternal );

   return (IInterface *)interfacePtr;
}
```

See Also: **SafInterface::interfaceType**

SafConnection::initiateConnection

```
SafStatus initiateConnection()
```

Description

This function opens a connection to a remote task. The name of the task to open a connection to is passed as parameter to the constructor function **SafConnection**. The method **establish()** is automatically called if it has not been called previously for this instance. This behaviour can be changed. Refer to the macro section.

The *SWBus* function **SbOpen()** is used to open the connection to the remote process

Note that the connection is not established before the virtual method **onConnect()** is called.

Access

This function is public.

Return Value

Return *Saf_SbIsConnected* if the task is connected to the remote process. If **establish()** fails the error code from this method is returned. Returns *Saf_SbConnectionFailed* if the

connection could not be opened. If the connection is successfully initiated *Saf_OK* is returned.

Example

This example demonstrates the use of **initiateConnection()** to establish a connection to a remote task.

```
void connect( Connection*& connection, const char* remoteTaskName )
{
  connection = new Connection( remoteTaskName );
  connection->initiateConnection();
}
```

See Also: SafConnection::establish, SafConnection::onConnect, SbOpen

SafConnection::internalInterfaceStore

`SafInterfaceStore& internalInterfaceStore()`

Description

This method gets a reference to the list of internal interfaces.

Access

This function is public.

Return Value

Returns a reference to the list of internal interfaces.

Example

Uses the methods **internalInterfaceStore()** and **remove()** to destroy an internal interface.

```
void Connection::destroyInterface( int32 iId )
{
   internalInterfaceStore().remove( iId, SafTTrue );
}
```

See Also: SafInterfaceStore

SafConnection::isConnected

`SafTBool isConnected() const`

Description

This method checks if the connection object is connected to the external task or not.

Access

This function is public.

Return Value

Returns *SafTTrue* if the object is connected to the external task, otherwise *SafTFalse*.

See Also: SafConnection::connectionIsIntiated

SafConnection::isEstablished

```
SafTBool isEstablished() const
```

Description

This methods verifies whether the method **establish()** has been activated or not.

Access

This function is public.

Return Value

IsEstablished() returns *SafTTrue* if the **establish()** method for this class has been called, otherwise *SafTFalse*.

See Also: SafConnection::establish

SafConnection::isNonBlocking

```
SafTBool isNonBlocking() const
```

Description

Check if non-blocking has been enabled for this connection. If a connection is closed and reopened again, the non-blocking mode is automatically reset.

Access

This function is public.

Return Value

isNonBlocking() returns *SafTTrue* if the connection is currently in non-blocking mode. If the connection to the external task is not established or the connection is not in non-blocking mode *SafTFalse* is returned.

See Also: SafConnection::enableNonBlocking

SafConnection::onConnect

```
virtual SafStatus onConnect()
```

Description

This virtual method is automatically invoked by *SAF* when the connection to the external task is established. A connection is not ready to be used before this method is called. The **onConnect()** method is called in both the client and the server application.

This method may be overloaded within a **SafConnection** sub-class. Normally this method should get *SWBus* indexes of functions and variables in the remote task.

Access

This is virtual protected method.

Return Value

The default implementation returns *Saf_OK*. The overloaded method must return *Saf_OK* on success, otherwise an error code.

Example

This example demonstrates a typical **onConnect**() method that uses the *SWBus* functions **SbIds()**, **SbHead()** and **SbTail()** to get indexes of functions in the remote process when the connection to the remote task is established. The class *MyConnection* is derived from **SafConnection**.

```
SafStatus MyConnection::onConnect()
{
   static const char* sFuncNames[] =
   {
      "RemoteFunction1",
      "RemoteFunction2",
      NULL
   };

   SbTSTI funcIds = SbIds( remoteSTI(), sFuncNames );
   if ( SbDBad( funcIds ) )
   {
      cout << "Failed to get STIs of functions" << endl;
      return Saf Error;
   }

   mFunc1 = SbHead( funcIds ); funcIds = SbTail( funcIds );
   mFunc2 = SbHead( funcIds );
   if ( SbDBad( mFunc1 ) || SbDBad( mFunc2 ) )
   {
      cout << "Failed to get function Ids." << endl;
      return Saf Error;
   }

   return Saf OK;
}
```

See Also: **SbIds, SbHead, SbTail, SafConnection::initiateConnection**

SafConnection::onConnecting

`virtual SafStatus onConnecting()`

Description

This virtual method is automatically invoked by *SAF* when a connection object has initiated a connection to a remote task. The task will periodically try to connect to the remote process at intervals specified by the environment variable *BUSRETRY*.

When this method is called the connection is not yet established.

This method may be overloaded in sub-classes of **SafConnection**.

Access

This is a virtual protected method.

Return Value

The default implementation returns *Saf_OK*. The overloaded method in sub-classes should return *Saf_OK* on success, otherwise an error status.

Example

```
SafStatus MyConnection::onConnecting()
{
   cout << "Trying to connect to process " << taskName()
        << endl;
   return Saf OK;
}
```

See Also: **SafConnection::onConnect, SafConnection::initiateConnection**

SafConnection::onConnectionBreak

`virtual SafStatus onConnectionBreak()`

Description

This virtual method is automatically invoked by *SAF* when the connection to the remote task is broken. Note that **onConnectionBreak()** is not called when the remote process terminates gracefully.

This method may be overloaded in sub-classes of **SafConnection**.

Access

This is a virtual protected method.

Return Value

The default implementation returns *Saf_OK*. The overloaded method in sub-classes should return *Saf_OK* on success, otherwise an error status.

Example

```
SafStatus MyConnection::onConnectionBreak()
{
   cout << "Connection to process " << taskName()
        << " broken" endl;

   return Saf OK;
}
```

See Also: **SafConnection::onConnect, SafConnection::onDisconnect**

SafConnection::onConnectionTimeout

`virtual SafStatus onConnectionTimeout()`

Description

This virtual method may be overloaded in sub-classes of **SafConnection**. Notice that this method is not automatically invoked by *SAF*. The method **SafTask::onConnectionTimeout()** is invoked when a process has given up establishing a connection to a remote process, because the period specified by the environment variable *BUSCONNECT* has elapsed. To call this method forward the call from **SafTask::onConnectionTimeout()**. See example.

For more information about this method refer to **SafTask::onConnectionTimeout()**.

Access

This is a public virtual method.

Return Value

The default implementation returns *Saf_OK*. The overloaded method in sub-classes should return *Saf_OK* on success, otherwise an error status.

Example

This example demonstrates how a **onConnectionTimeout()** method may be implemented in a *SAF* compliant application. The class *MyConnection* is a sub-class of **SafConnection**, while *MyTask* is a sub-class of **SafTask**. Notice that the task forwards the call to the connection object.

```
SafStatus MyConnection::onConnectionTimeout()
{
   cout << "Timeout to task " << taskName() << endl;
   return Saf OK;
}


SafStatus MyTask::onConnectionTimeout( const char* taskName )
{
   SafConnection* c = findConnection( taskName );
   if ( c )
     return c->onConnectionTimeout();

   return Saf Error;   // Not found.
}
```

See Also: **SafConnection::onConnect, SafConnection::initiateConnection, SafTask::onConnectionTimeout**

SafConnection::onDisconnect

`virtual SafStatus onDisconnect()`

Description

This virtual method is automatically invoked by *SAF* when the connection to the remote task is terminated gracefully. Note that **onDisconnect()** is not called when the connection to the remote task is broken.

This method may be overloaded in sub-classes of **SafConnection**.

Access

This is a virtual protected method.

Return Value

The default implementation returns *Saf_OK*. The overloaded method in sub-classes should return *Saf_OK* on success, otherwise an error status.

Example

```
SafStatus MyConnection::onDisconnect()
{
   cout << "Connection to process " << taskName()
        << " disconnected" endl;

   return Saf OK;
}
```

See Also: **SafConnection::onConnect, SafConnection::ConnectionBreak**

SafConnection::onIncompatibleVersions

```
vitual SafStatus onIncompatibleVersions()
```

Description

This virtual method may be overloaded in sub-classes of **SafConnection**. Notice that this method is not automatically invoked by *SAF*. The method **SafTask::onIncompatibleVersions()** is invoked when the process trying to establish a connection to where incompatible with the *SWBus* version of the local task. To call this method forward the call from **SafTask::onIncompatibleVersions()**. See example.

For more information about this method refer to **SafTask::onIncompatibleVersions()**.

Access

This is a virtual public method.

Return Value

The default implementation returns *Saf_OK*. The overloaded method in sub-classes should return *Saf_OK* on success, otherwise an error status.

Example

This example demonstrates how a **onIncompatibleVersions()** method may be implemented in a *SAF* compliant application. The class *MyConnection* is a sub-class of **SafConnection**, while *MyTask* is a sub-class of **SafTask**. Notice that the task forwards the call to the connection object.

```
SafStatus MyConnection::onIncompatibleVersions()
{
   cout << "Process " << taskName()
        << " is incompatible with this process" << endl;
   return Saf OK;
}


SafStatus MyTask:: onIncompatibleVersions ( const char* taskName )
{
   SafConnection* c = findConnection( taskName );
   if ( c )
     return c-> onIncompatibleVersions();

   return Saf Error;   // Not found.
}
```

See Also: **SafConnection::initiateConnection, SafTask::onIncompatibleVersions**

SafConnection::onVariableUpdate

```
virtual SafStatus onVariableUpdate( int32     numVariables,
                                    SbTSTI* variables )
```

Description

This virtual method may be called when an external task has sent updated variables to the local task. The default implementation of this method only returns *Saf_OK*. This function may be overloaded in sub-classes of **SafConnection**.

Note that this method is not automatically invoked by *SAF*. When **SbFlush()** is invoked in the external task *SAF* will automatically call the method **SafTask::onVariableUpdate()**. The

example below demonstrates how to forward the call to the **SafConnection** method **onVariableUpdate()**. Refer to the example.

Access

This is a virtual public method.

Parameters

numVariables Number of updated variables.

variables Array of *SWBus STI*s of the updated variables.

Return Value

The default implementation returns *Saf_OK*. The overloaded method in sub-classes of **SafConnection** should return *Saf_OK* on success, otherwise an error status.

Example

This example demonstrates how to forward the call **onVariableUpdate()** from a **SafTask** sub-class to a **SafConnection** sub-class. Note that the implementation of the *update()* method is skipped in this example.

```
SafStatus Connection::onVariableUpdate( int32   numVariables,
                                        SbTSTI* variables )
{
   for ( int i = 0; i < numVariables; i++ )
      update( SbName(  variables[i] ),
              SbClass( variables[i] ),
              SbData(  variables[i] ) );
   return Saf OK;
}

SafStatus Task::onVariableUpdate( const char* taskName,
                                  int32       numVariables,
                                  SbTSTI*     variables )
{
    SafConnection* c = findConnection( taskName );
    if ( c )
       return c->onVariableUpdate( numVariables, variables );

    return Saf Error;
}
```

See Also: SafTask::onVariableUpdate, SbClass, SbData

SafConnection::remoteSTI

`SbTSTI remoteSTI() const`

Description

Returns the *SWBus STI* for the task's local representation of the remote process.

Access

This function is public.

Return Value

The *SWBus STI* of the remote process. If the connection is not established *SbCNull* is returned.

See Also: SafConnection::initiateConnection

SafConnection::SafConnection

SafConnection(const char* *taskName* **)**

Description

This method is the constructor for the **SafConnection** class. It initialises the connection and registers itself in the **SafTask** object instance. The task name must be specified. An error message is given if the supplied task name is not recognized as a valid task name.

Access

This constructor is public.

Parameters

taskName Name of task to establish a connection to.

Example

This example demonstrates how a client creates a connection to a server named MyServer. *ServerConn* is a sub-class of **SafConnection**.

```
main()
{
   …
   ServerConn connection( "MyServer" );
   if ( connection.initiateConnection() != Saf OK )
      return 1;
   …
}
```

See Also: SafConnection::initiateConnection, SafConnection::~SafConnection, SafTask

SafConnection::sub

static SbTSTI sub(const char* *name* **)**

Description

This static method creates a new *SWBus* class called *name*. The new class will be a sub-class of an internal *SAF* class. Instances of classes created using **sub()** must be used as input parameter to the method **call()**. Use **SbAdd()** to add fields to this class.

The *SWBus* class created using **sub()** and the corresponding *C* struct must have the same layout. It is extremely important that the first field in the *C* struct is the macro **SafDConnectionInfo** or that it is derived directly from **SafConnectionClassStruct**.

Whencalling a connection object method in the remote process always use the methods **declare()**, **sub()** and **call()**. Do not mix these functions with a direct call to **SbDeclare()**, **SbSub()** or **SbCall()**. It is extremely important to be aware of this because the function **SafFGetConnection()** can not be used to get correct connection object in the remote process if these functions are mixed. The exception is the output parameter. It is not necessary to use the method **sub()** to create the *SWBus* class because no extra information is used when returning from the *SWBus* function.

Access

This is a static public function. It can be accessed without having an instance of a **SafConnection** sub-class.

Parameters

name Name of the new *SWBus* class.

Return Value

Returns the *SWBus STI* of the new class, or *SbCError* if the operation failed.

Example

This example defines an *SWBus* class called *mMyClass* and a corresponding *C* struct named *sMyClass*.

```
typedef struct
{
   SafDConnectionInfo; // Must be the first element
   char* string;
} sMyClass;


SafStatus MyConnection::create()
{
   mMyClass = sub( "mMyClass" ); // struct type
   if ( SbDBad( mMyClass ) )
   {
     cout << "Failed to create class" << endl;
     return Saf Error;
   }

   SbTSTI field = SbAdd( mMyClass, SbCCString, "string", 0 );
   if ( SbDBad( field ) )
   {
      cout << "Failed to add field to class" << endl;
      return Saf Error;
   }

   return Saf OK;
}
```

See Also: SafConnection::call, SafConnection::declare, SbSub, SbAdd

SafConnection::taskClass

`const char* taskClass() const`

Description

Returns the task class for the remote task.

Access

This function is public.

Return Value

If the connection is established the task class for the remote task is returned, otherwise NULL.

Example

```
void SafConnection::check()
{
   if ( isConnected() )
      cout << "TaskClass = " << taskClass() << endl;
}
```

See Also: SafConnection::isConnected

SafConnection::taskName

```
const char* taskName() const
```

Description

Returns the name of the remote task to establish connection to or is already connected to.

Access

This function is public.

Return Value

Returns the name of the remote task.

See Also: **SafConnection::SafConnection**

SafConnection::~SafConnection

```
~SafConnection()
```

Description

This method is the destructor for the **SafConnection** class. Deletes all interface objects associated with this instance and removes the connection object reference from the **SafTask** object instance.

Access

This destructor is public.

See Also: **SafConnection::SafConnection, SafTask**

Macros

SafDConnectionInfo

```
SafDConnectionInfo
```

Description

This macro is used when creating a *C* struct corresponding to an *SWBus* class created using the **SafConnection** method **sub()**. It is very important that this macro is put first in the struct definition otherwise undefined behaviour may be the result.

It is also possible to define a struct deriving directly from **SafConnectionClassStruct**.

Example

```
struct sIn1
{
   SafDConnectionInfo;
   …           /* Add fields */
};

struct sIn2 : public SafConnectionClassStruct
{
   …            /* Add fields */
};
```

See Also: **SafConnection::sub, SafConnection::call**

SafDDeclareConnection

`SafDDeclareConnection`

Description

This macro together with **SafDImplementConnection** is used to prevent that the **establish()** method is called more than one time for a specific **SafConnection** sub-class. When these macros are used only the first instance created of a sub-class of **SafConnection** will call the **establish()** method.

This macro must be put inside the class definition. Normally this class is defined in a header file.

Note that **SafDImplementConnection** must be added outside the class definition if this macro is used. If not the compiler will produce several error messages as a result.

Example

This is an example that defines a connection class using the class **SafConnection** as a base class. It adds the **SafDDeclareConnection** macro. This is a typical implementation of a **SafConnection** sub-class. It is defined in a header file.

```
class Connection : public SafConnection
{
 private:
    …
    SafStatus establish();

 public:
    Connection( const char* taskName );
    ~Connection();

    …

 SafDDeclareConnection;
};
```

The implementation continues below in the *C* file. The macro **SafDImplementConnection()** is used to define the static variables declared in the macro **SafDDeclareConnection**. Note the use of the class name as argument.

```
SafDImplmentConnection( Connection );

Connection::Connection( const char* taskName ) :
   SafConnection( taskName )
{
  …
}

Connection::~Connection()
{
   …
 }

SafStatus Connection::establish()
{
  …
  return Saf OK;
}
```

See Also: **SafConnection::establish, SafDImplementConnection, SafDDestroyConnection**

SafDDestroyConnection

`SafDDestroyConnection`

Description

This macro is used to call the virtual method **destroy()** from the destructor of a **SafConnection** sub-class. Put this macro into the destructor of the sub-class.

If using the macros **SafDDeclareConnection** and **SafDImplementConnection** the method **destroy()** will only be called when deleting the last instance of a **SafConnection** sub-class. If these macros are not used it will be called each time an instance is deleted.

Example

```
Connection::~Connection()
{
   …
   SafDDestroyConnection;
}


SafStatus Connection::destroy()
{
   …
   return Saf OK;
}
```

See Also: **SafConnection::~SafConnection, SafConnection::destroy, SafDDeclareConnection, SafDImplementConnection**

SafDImplementConnection

`SafDImplementConnection(` *SAFCLASS* **`)`**

Description

This macro is used together with the macro **SafDDeclareConnection** to prevent that the **SafConnection** sub-class method **establish()** is called more than one time for a given class. For more information refer to **SafDDeclareConnection**.

Parameters

SAFCLASS *Name of a **SafConnection** sub-class.*

Example

See example for **SafDDeclareConnection**.

See Also: **SafImplementConnection**

Global Functions

SafFGetConnection

`SafConnection* SafFGetConnection( SbTSTI` *in* **`)`**

Description

This function returns a pointer to a **SafConnection** instance based on the information passed in the parameter *in*. The parameter *in* must be created in the remote process using the **SafConnection** method **sub()**.

Calling the **SafConnection** method **call()** without an input argument still creates enough information for **SafFGetConnection()** to get proper **SafConnection** instance. Refer to **sub()** and **call()** for a detailed description.

This function should only be called from an *SWBus* function that was invoked from a remote process using the **SafConnection** method **call()**.

Parameters

in Input parameter passed to the *SWBus* function.

Return Value

Returns a pointer to a **SafConnection** object if it is found in *SAF*, otherwise *NULL*.

Example

This example demonstrates a typical implementation of an *SWBus* function that calls a method in a **SafConnection** object.

```
SbTSTI restoreCB( SbTSTI, SbTSTI, SbTSTI in, SbTSTI out )
{
    Connection* c = (Connection *)SafFGetConnection( in );
    if ( c )
       c->restore( in );
    else
       cout << "Failed to find connection object" << endl;

    return out;
}
```

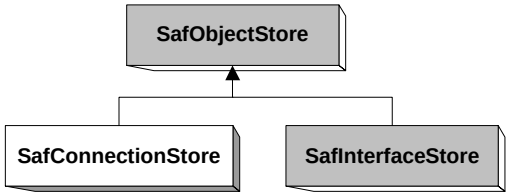
See Also: SafConnection::sub, SafConnection::call, SafFGetInterface

SafConnectionStore

A **SafConnectionStore** object maintains an ordered collection of any object derived from the class **SafConnection**. The objects are put into a double linked list.

SafConnection objects can be added and removed from the list. The list can be traversed.

This class may be sub-classed further to get extended functionality present in the generic base class **SafObjectStore**.



Class Members

Construction/Destruction	
~SafConnectionStore	Destroys **SafConnectionStore** object.

Operations	
append	Appends a new connection object to the list.
doDelete	Deletes a connection object.
remove	Removes a connection object from the list.

Get Methods	
contains	Checks if a connection object is in the list.
getNext	Used to get the next connection object in the list.
isEqual	Checks if two connection objects are equal.

Member Functions

SafConnectionStore::append

`SafTBool append( const SafConnection*` *conn* **`)`**

Description

Appends the **SafConnection** sub-class object passed as argument to the end of the list.

Access

This function is public.

Parameters

conn **SafConnection** sub-class instance to add to the list.

Return Value

Returns *SafTTrue* if the connection object is successfully appended to the list. If the operation failed *SafTFalse* is returned.

See Also: SafConnectionStore::remove

SafConnectionStore::contains

```
SafConnection* contains( const char* name ) const
```

Description

Traverses the list and returns the **SafConnection** sub-class instance with the name of the connection equal to the parameter *name*. If the *name* does not exist in the list NULL is returned.

Access

This function is public.

Parameters

name Name of the connection object to find in the list.

Return Value

Returns a pointer to the **SafConnection** sub-class instance if the object is found, otherwise NULL.

See Also: SafConnectionStore::append, SafConnectionStore::remove

SafConnectionStore::doDelete

```
void doDelete( void* object )
```

Description

This protected method deletes the **SafConnection** sub-class instance. The parameter *object* points to the connection object to delete. The instance is deleted using the *C++* operator *delete*.

This method is called from within the **SafConnectionStore** and **SafObjectStore** methods. Note that this method is only of interest to developers who sub-classes **SafConnectionStore**.

Access

This function is protected. Can only be called from sub-classes of **SafConnectionStore**.

Parameters

object The instance to delete.

See Also: SafConnectionStore::remove, SafConnectionStore::isEqual

SafConnectionStore::getNext

SafConnection* getNext(SafPOSITION& *pos*) const

Description

This method is used when traversing the **SafConnection** sub-class objects in the list. It returns a pointer to current connection object. The parameter *pos* is set to point to the next connection object in the list.

To get the first object in the list the parameter *pos* must be initialised by calling the base class method **getHeadPosition()**.

Access

This function is public.

Parameters

pos	A reference to the position of the connection object to return. After the function call the position is pointing to the next connection object. When reaching the end of the list the value of this variable is NULL. Use NULL as a termination condition in a loop.

Return Value

Returns a pointer to the **SafConnection** sub-class object.

Example

This example demonstrates how to traverse a list of **SafConnection** objects. The class *TheTask* is derived from **SafTask**.

```
void TheTask::printConnectionClasses( SafConnectionStore& list )
{
   SafPOSITION pos  = list.getHeadPosition();
   while ( pos )
   {
      SafConnection* conn = list.getNext( pos );
      cout << "Connection class = " << conn->taskClass() << endl;
   }
}
```

See Also: **SafObjectStore::getHeadPosition**

SafConnectionStore::isEqual

SafTBool isEqual(const void* *o1*, const void* *o2*) const

Description

Checks if the objects *o1* and *o2* are equal.

This method is called from **SafConnectionStore** and **SafObjectStore** methods. Note that this method is only of interest to developers who sub-classes **SafConnectionStore**.

Access

This function is protected. Can only be called from sub-classes of **SafConnectionStore**.

Parameters

o1	Pointer to connection object 1.

o2 Pointer to connection object 2.

Return Value

Returns *SafTTrue* if the objects are equal, otherwise *SafTFalse*.

See Also: SafConnectionStore::doDelete

SafConnectionStore::remove

```
SafTBool remove( const SafConnection* conn,
                 SafTBool deletElement = SafTTrue )
```

Description

This method removes the **SafConnection** sub-class object passed as argument from the list. If the parameter *deleteElement* is true the connection object is deleted.

Access

This function is public.

Parameters

conn Pointer to the connection object to remove from the list.

deleteElement The connection object is deleted if set to *SafTTrue*. The object will not be deleted if set to *SafTFalse*.

Return Value

Returns *SafTTrue* if the object is successfully removed from the list, or *SafTFalse* if the operation failed.

See Also: SafConnectionStore::append

SafConnectionStore::~SafConnectionStore

```
~SafConnectionStore()
```

Description

This method is the destructor for the class **SafConnectionStore**. It removes all the objects from the list. Note that the objects in the list are not deleted.

Access

This destructor is public.

Example

This example demonstrates how to delete all the objects in the list. The base class method **removeAll()** is used to delete the connection objects.

```
void remove( SafConnectionStore* connStore )
{
   // Delete all connection objects
   connStore->removeAll( SafTTrue );
   delete connStore;
}
```

See Also: SafObjectStore::removeAll

SafContext

SafContext is a struct used in the external interface **SafReportExternalInterface**. It contains information about a context in the remote task.

The information stored in a **SafContext** can be used to enable or disable information printed to an output device in a remote application.

The following contexts are predefined in *SAF*:

- **SAF_CON_SWBUS**
- **SAF_CON**
- **SAF_CON_FLOW**

For more information about how to create new contexts refer to the **SafReport** macro section.

Refer to the **SafReportExternalInterface** description for more information about why and how this struct can be used.

Struct Methods

Attributes	
mDescription	Description of the context.
mName	Human readable name for this context.
mString	Unique context identifier.

Data Members

SafContext::mDescription

Description

This is the description of the context. For more information about this attribute refer to the **SafReport** macro section.

SafContext::mName

Description

This is the human readable name for the context. This name does not need to be unique. For more information about this attribute refer to the **SafReport** macro section.

SafContext::mString

Description

This attribute is the name of the context used when printing a message. It must be a unique name. This is also the name of the context variable in the remote task. For more information about this attribute refer to the **SafReport** macro section.

SafExternalInterface

A **SafExternalInterface** sub-class instance is a client's interface to a server program. It is used by a client to access functionality offered by a server.

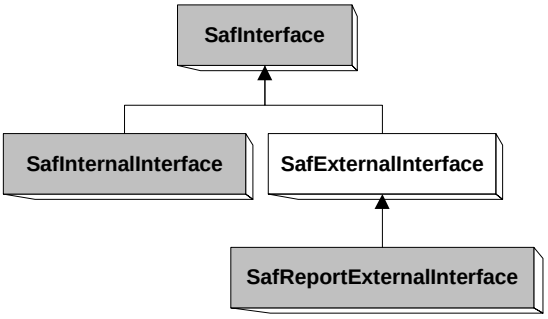
An external interface should hide *SWBus* functionality from a client program and should be developed by the same development team that created the server module. A client program will only use methods offered by an external interface to access functionality in the server. When a client creates and initialises an external interface, a corresponding internal interface will be created in the server process. These objects then have a logical connection. When a client calls a method in an external interface, a corresponding method will be called in an internal interface in the server process.

SafExternalInterface must be further sub-classed and functionality specific for the interface must be added. A client program may have several interfaces. These interfaces can be instances of the same class or from completely different classes with different functionality. **SafExternalInterface** is derived from **SafInterface.**

When creating an external interface an interface class must be supplied. This interface class must be consistent with the interface class expected within the server program. This is extremely important because this interface class is the only name that a server can use to distinguish between the different internal interfaces it can create. This name must be decided and defined at design time and it must be unique. The parameter *interfaceClass* in the class **SafExternalInterface** constructor should be hard coded by the developer of the external interface. When designing an external interface the user must never get the opportunity to change the interface class because it may result in unpredictable behaviour. This interface class must be known at design time in the server module. Note that the method **createInterface()** is called in the server module to create an internal interface based on information sent when a client creates an external interface.

SAF offers macros to simplify the process of creating several interfaces of a class. Refer to the **SafInterface** macro section for further details.

For a detailed description of the base class members refer to **SafInterface**.



Class Members

Construction/Destruction

SafExternalInterface	Constructs **SafExternalInterface** object.
~SafExternalInterface	Destroys **SafExternalInterface** object.

Overloaded Methods

interfaceType	Returns *SafCExternal.*
initialize	Initialises the object and creates internal interface in the server.

Member Functions

SafExternalInterface::interfaceType

```
int32 interfaceType() const
```

Description

This is a pure virtual method in **SafInterface** that is implemented in this class. Returns *SafCExternal*.

Access

This function is public.

Return Value

Returns SafCExternal.

See Also: SafInterface::interfaceType

SafExternalInterface::initialize

```
SafStatus initialize()
```

Description

This method initialises the external interface object. It calls the virtual method **establish()** used for creating *SWBus* objects. This behaviour can be changed. Refer to the **SafInterface** macro section for details. If the method **establish()** returned *Saf_OK* it continues in the function and if the connection to the remote task is established it calls the **SafConnection** sub-class method **createInterface()** to create the associated internal interface. When the internal interface is created and initialised in the remote process the virtual method **onConnect()** will be called.

If the connection to the remote task is not established when calling the method **initialize()** it will automatically call the method **createInterface()** in the **SafConnection** sub-class when the connection is established.

Access

This function is public.

Return Value

If the connection to the remote task is not connected *Saf_SbNotConnected* is returned if the **establish()** method returned *Saf_OK*, otherwise the error code from **establish()** is returned.

If the connection to the remote task is established the return value from **onConnect()** is returned, otherwise the error status from the **establish()** method is returned.

Example

Refer to the example demonstrated for SafExternalInterface::SafExternalInterface.

See Also: SafExternalInterface::SafExternalInterface, SafInterface::initialize

SafExternalInterface::SafExternalInterface

```
SafExternalInterface( SafConnection* connection,
                      const char*    interfaceClass )
```

Description

This method is the constructor for the **SafExternalInterface** class. It initialises the external interface object. The arguments *connection* and *interfaceClass* are passed directly to the base class **SafInterface**.

The base class **SafInterface** is initialised with a unique interface identifier that is created in the **SafExternalInterface** constructor. This interface identifier must be passed to the associated **SafInternalInterface** object in the server task. It is passed as argument to the **SafConnection** method **createInterface()**.

The **SafExternalInterface** instance is registered with the **SafConnection** instance that is passed as argument to the constructor.

Access

This constructor is public.

Parameters

connection	Pointer to a **SafConnection** instance.
interfaceClass	Name of the interface class. This *interfaceClass* will be passed as argument to the **SafConnection::createInterface()** in the server task.

Example

This example demonstrates the use of **SafConnection** and **SafExternalInterface** sub-classes that creates and initialises an external interface.

```
void create()
{
  …
  MyExternalInterface* eiPnt;
  MyConnection*        connectionPnt;

              // Connect to the task MyServer
  connectionPnt= new MyConnection( “MyServer” );
  connectionPnt->initiateConnection();

  eiPnt = new MyExternalInterface( connection, "MonitorEI" );
  eiPnt->initialize();
  …
}
```

See Also: **SafExternalInterface::~SafExternalInterface, SafInterface::SafInterface, SafConnection**

SafExternalInterface::~SafExternalInterface

```
~SafExternalInterface()
```

Description

Destroys the **SafExternalInterface** object and removes the interface object from the connection object’s interface list.

Access

This destructor is public.

See Also: **SafExternalInterface::SafExternalInterface, SafInterface::SafInterface, SafInterface::~SafInterface**

SafHandler

This class is an abstract class and is used as a base class for the classes **SafIO** and **SafPeriodic**. It should not be necessary to sub-class **SafHandler** directly in a *SAF* application. **SafHandler** is a base class implementing functionality used in the classes **SafIO** and **SafPeriodic**. A *SAF* application will sub-class **SafIO** and **SafPeriodic**. Sub-classes of **SafIO** are used when listening for input on a *SOCKET* or a file descriptor on *UNIX*. Sub-classes of **SafPeriodic** are used for creating objects to be activated at regular intervals.

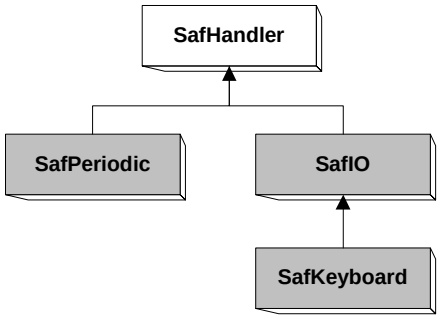
Sub-classes of **SafHandler** must implement the pure-virtual method **construct()**.

The following virtual methods can be overloaded in user-defined sub-classes of **SafIO** and **SafPeriodic**:

- **onAction**
- **onSuspend**
- **onRestart**
- **onActivate**

Functionality for creating, activating, suspending and restarting handler objects are implemented in this class. In other words generic handler functionality is put into the class **SafHandler**.

Specific handler functionality, different for the different kinds of handlers, is implemented in the existing **SafHandler** sub-classes. For more information about these classes see description for **SafIO** and **SafPeriodic**.



Class Members

Construction/Destruction	
SafHandler	Constructs **SafHandler** object.
~SafHandler	Destroys **SafHandler** object.
construct	Constructs the *SWBus* handler object.

Overridable Methods	
onAction	Called in response to activity on the object.
onActivate	Called when the object is activated.
onRestart	Called when the object is restarted.
onSuspend	Called when the object is suspended.

Get Methods	
isActivated	Checks if the object is activated.
isRunning	Checks if the object is running.
isSuspended	Checks if the objects is suspended.

Operations	
activate	Activates the object.
restart	Restarts the object.
suspend	Suspends the object.

Pure Methods	
construct	Must construct the *SWBus* object and call the **SafHandler::construct** method.

Attributes	
mSTI	The *SWBus STI* of the handler object.
mState	The running state of the object.

Member Functions

SafHandler::activate

```
SafStatus activate()
```

Description

This method is called to activate the **SafHandler** sub-class object. After this method is called the object will be in a running state.

It checks if the handler has already been restarted. If not, it checks if the handler is currently suspended. In that case it is just restarted. If it is has not been suspended and not already been activated, the **onActivate()** method is called. If the activation of this handler succeeds, this method finally calls the overloaded method of the **construct()** method in the **SafHandler** sub-class.

Access

This function is public.

Return Value

The return value of this method depends on the running state of the handler object.

If the state is running *Saf_OK* is returned.

If the state is suspended the return value from **restart()** is returned.

If the state is not activated the error code from **onActivate()** is returned if it failed, otherwise the value from the **construct()** method is returned.

If the state is activated and not running *Saf_InternalError* is returned.

See Also: **SafHandler::construct, SafHandler::onActivate, SafHandler::restart, SafHandler::suspend**

SafHandler::construct

```
SafStatus construct( SbTSTI theClass,
                     SbTSTI constrParam )

virtual SafStatus construct() =0
```

Description

The virtual **construct()** method is pure virtual and must be overloaded in sub-classes of **SafHandler**. It must create the constructor parameter used when calling the non-virtual **construct()** method. The *constrParam* must be either *SbCCIoParam* or *SbCCPeriodicParam*. The object must also be set in a running state.

The non-virtual **construct()** method creates the *SWBus* handler object. The parameter *theClass* must be either *SbCCIo* or *SbCCPeriodic.* These are the only *SWBus* classes supported in the current version of *SAF*. This method should not be called in any circumstances from other methods than the overloaded **construct()** method in sub-classes of **SafHandler**.

It is recommended that the classes **SafIO** and **SafPeriodic** are used as base class instead of sub-classing **SafHandler** directly.

Access

Both functions are protected. Can only be called from sub-classes of **SafHandler**.

Parameters

theClass	Must be one of the predefined *SWBus* classes *SbCCIo* or *SbCCPeriodic*.
constrParam	Constructor parameter. Must be either *SbCCIoParam* or *SbCCPeriodicParam*.

Return Value

The virtual **construct()** method must return *Saf_OK* on success, otherwise an error code.

The non-virtual **construct()** method returns *Saf_OK* on success. If the parameter *theClass* is not of the expected type *Saf_SbIllegalArgs* is returned. *Saf_SbCreateFailed* is returned if the function failed to create the *SWBus* object.

Example

This example demonstrates how to implement an IO handler that is activated in response to activity on a *SOCKET*. The class *myIOClass* is derived from **SafHandler**.

```
SafStatus myIOClass::construct()
{
   SbTIoParameter constrParamData;

   constrParamData.fdc = SbCFDREAD;
   constrParamData.fds = &mFds;

   SbTSTI constrParam = SbCreate( SbCPLocal, SbCCIoParameter,
                                  NULL, SbCNull,
                                  &constrParamData );

   SafStatus sts = SafHandler::construct( SbCCIo, constrParam );
   SbDelete( constrParam, SbCNull );

   if ( sts == Saf OK )
      mState = eRunning;

   return sts;
}
```

See Also: **SafIO, SafPeriodic, SafHandler::mState**

SafHandler::isActivated

`SafTBool isActivated() const`

Description

Checks if the object is activated. An object is activated if the **activate()** method has been successfully called.

Access

This function is public.

Return Value

Returns *SafTTrue* if the object has been activated, otherwise *SafTFalse*.

See Also: **SafHandler::activate**

SafHandler::isRunning

`SafTBool isRunning() const`

Description

Checks if the object is in a running state. An object is running if **activate()** or **restart()** has been successfully called.

Access

This function is public.

Return Value

Returns *SafTTrue* if the object is running, otherwise *SafTFalse*.

See Also: **SafHandler::activate, SafHandler::restart**

SafHandler::isSuspended

`SafTBool isSuspended() const`

Description

Checks if the object is suspended. An object is suspended if the **suspend()** method has been successfully called.

Access

This function is public.

Return Value

Returns *SafTTrue* if the object is suspended, otherwise *SafTFalse*.

See Also: **SafHandler::suspend**

SafHandler::onAction

`virtual SafStatus onAction()`

Description

This method is called either as a response to activity on a *SOCKET* or file descriptor on *UNIX* or when the time has elapsed for a periodic object.

Normally this method will be overloaded in sub-classes of **SafIO** or **SafPeriodic**.

The default implementation of this method only returns *Saf_OK*.

Access

This is a virtual protected method.

Return Value

Should return *Saf_OK* on success, otherwise an error code.

Example

This example demonstrates how to implement an **onAction**() method that is a member of a class that is derived from **SafIO**. When input on the *SOCKET* is received the message is read. The example is made as simple as possible. No error checking is made.

```
SafStatus IOClass::onAction()
{
   static char buffer[65535];

   recv( TheSocket, buffer, 65535, 0 );
   update( buffer );   // Do something with the data received

   return Saf OK;
}
```

See Also: **SafHandler::activate**

SafHandler::onActivate

```
virtual SafStatus onActivate()
```

Description

This virtual method is called when the object is being activated. It is called before the *SWBus* object has been created.

This method may be overloaded in sub-classes.

Access

This is a virtual protected method.

Return Value

Must return *Saf_OK* on success, otherwise an error code.

See Also: SafHandler::activate, SafHandler::construct

SafHandler::onRestart

```
virtual SafStatus onRestart()
```

Description

This virtual method is called when the object is restarted.

This method may be overloaded in sub-classes for instance to initialise data on every restart.

Access

This is a virtual protected method.

Return Value

Must return *Saf_OK* on success, otherwise an error code.

See Also: SafHandler::restart

SafHandler::onSuspend

```
virtual SafStatus onSuspend()
```

Description

This virtual method is called when the object is suspended.

This method may be overloaded in sub-classes for instance to reset data when the object has been suspended.

Access

This is a virtual protected method.

Return Value

Must return *Saf_OK* on success, otherwise an error code.

See Also: SafHandler::suspend

SafHandler::restart

```
SafStatus restart()
```

Description

This method restarts the object and activates the virtual method **onRestart()**.

If the object is running when this method is invoked it only returns.

Access

This function is public.

Return Value

Returns *Saf_OK* if the object is successfully restarted. If the object is not suspended *Saf_SbNotSuspended* is returned. *Saf_SbRestartFailed* is returned if the object failed to be restarted.

See Also: SafHandler::onRestart

SafHandler::suspend

```
SafStatus suspend()
```

Description

If the object is running this method suspends the object and calls the **onSuspend()** virtual method.

If the object is in a suspended state when this method is invoked it only returns.

Access

This function is public.

Return Value

Returns *Saf_OK* if the object is successfully suspended or it is in a suspended state. If the object failed to be suspend *Saf_SbSuspendFailed* is returned.

See Also: SafHandler::onSuspend

SafHandler::SafHandler

```
SafHandler()
```

Description

This method is the constructor for the class **SafHandler**. It initialises the member attributes.

Access

This constructor is public.

See Also: SafHandler::~SafHandler

SafHandler::~SafHandler

```
virtual ~SafHandler()
```

Description

This method is the destructor for the class **SafHandler**. It deletes the *SWBus* object.

Access

This destructor is public.

See Also: **SafHandler::SafHandler**

Data Members

SafHandler::mState

Description

The attribute **mState** indicates the **SafHandler** object's current status. This attribute can have one of the following values:

- eCreated Indicates that the object is initialised.
- *eActivated* Indicates that the object has been created and activated.
- *eSuspended* Indicates that the object has been suspended.
- *eRunning* Indicates that the object is running.

Access

This attribute is protected.

SafHandler::mSTI

Description

The *SWBus STI* of the created *SWBus* object.

Access

This attribute is protected.

SafHashTable

The class **SafHashTable** is used as a container for a hash-table of generic data objects. A hash-table is a dictionary collection that maps unique keys to values. The data objects can be of any type.

This is an abstract class and must be further sub-classed to add functionality for objects of a specific class. It is not a template class hence only generic *void** are used as arguments to class members. Thus new methods should be created in sub-classes of **SafHashTable** using the generic base class methods.

It provides functionality for adding, removing and getting data objects from the hash-table.

The following methods must be overloaded in sub-classes:

- **hash**
- **isEqual**
- **doDelete**

The example code used throughout the description of **SafHashTable** is based on the class *VariableMap* that derives from **SafHashTable**. The objects inserted into the *VariableMap* instance are of type *VariableObject*. The class definition for *VariableObject* is not of interest in the following example code and is therefore skipped. The hash key used in the example code is of type integer. The data type integer is used for simplicity but since this class is generic any data type can be used as key.

Class Members

Construction/Destruction

SafHashTable	Constructs **SafHashTable** object.
~SafHashTable	Destroys **SafHashTable** object.

Pure Overridable Methods

doDelete	Must delete the object passed as argument.
hash	Must return the index in the hash-table where to put the object.
isEqual	Checks if two objects are equal.

Operations

doAdd	Adds a new object to the hash-table.
doRemove	Removes an object from the hash-table.
rehash	Must return the index in the hash-table where to put the object when the hash-table is resized.
removeAll	Removes all the objects from the hash-table.
resize	Resizes the hash-table.

Get Methods	
doLookup	Looks up the object mapped to a given key.
itContains	Checks if an object for a given key exists in the hash-table.
tableSize	Number of buckets in the hash-table.

Member Functions

SafHashTable::doAdd

SafTBool doAdd(void* *hObject*, **void*** *object* **)**

Description

This method inserts a new object into the hash-table. First the key is looked up using the parameter *hObject* as argument to the virtual method **hash()**. The method **hash()** returns an index in the hash-table. Secondly, the parameter *object* is inserted into the hash-table at the given index.

Access

This function is protected. Can only be called from sub-classes of **SafHashTable**.

Parameters

hObject	Pointer to the hash object key.
object	Pointer to the object to put into the list.

Return Value

Returns *SafTTrue* if the object was successfully inserted into the hash-table, or *SafTFalse* if the operation failed.

Example

```
SafTBool VariableMap::add( int key, VariableObject* object )
{
   return doAdd( (void *)key, (void *)object );
}
```

See Also: **SafHashTable::doRemove, SafHashTable::doDelete, SafHashTable::hash**

SafHashTable::doDelete

virtual void doDelete(void* *object* **) =0**

Description

This pure virtual method must delete the object passed as argument. The class **SafHashTable** must be further sub-classed and this method must be overloaded.

If the object was allocated using *new* the same object must be destroyed using *delete*. Note that the object must be cast to the correct class before it is deleted.

Access

This is a pure virtual protected method. Must be implemented in sub-classes of **SafHashTable**.

Parameters

object Pointer to the object to delete.

Example

```
void VariableMap::doDelete( void* object )
{
   delete (VariableObject *)object;
}
```

See Also: SafHashTable::doAdd

SafHashTable::doLookup

void* doLookup(void* *hObject* **) const**

Description

This method uses a hashing algorithm to quickly find the map object with a key that exactly matches the given key. The virtual methods **hash()** and **isEqual()** are used to find the object.

Access

This function is protected. Can only be called from sub-classes of **SafHashTable**.

Parameters

hObject Pointer to the hash key object.

Return Value

Returns a pointer to the object if it was found in the hash-table, otherwise NULL.

Example

```
VariableObject* VariableMap::lookup( int key ) const
{
   return (VariableObject *)doLookup( (void *)key );
}
```

See Also: SafHashTable::hash, SafHashTable::isEqual, SafHashTable::itContains

SafHashTable::doRemove

void* doRemove(void* *hObject* **)**

Description

This method looks up the object corresponding to the supplied key *hObject* and removes it from the hash-table if it was found. The method does not delete the object, but returns a pointer to it.

The virtual methods **hash()** and **isEqual()** are used to find the object.

Access

This function is protected. Can only be called from sub-classes of **SafHashTable**.

Parameters

hObject Pointer to the hash key object.

Return Value

Returns a pointer to the object if it was found in the hash-table, otherwise NULL.

Example

```
void VariableMap::remove( int key )
{
    VariableObject* object = doRemove( (void *)key );
    if ( object )
       delete object;
}
```

See Also: **SafHashTable::removeAll, SafHashTable::hash, SafHashTable::isEqual**

SafHashTable::hash

virtual uint32 hash(void* *hObject* **) const =0**

Description

This pure virtual method must return an index within the hash-table. The parameter *hObject* is the hash key object. Based on this argument an index must be calculated.

The class **SafHashTable** must be further sub-classed and this method must be overloaded.

The size of the hash-table is initially set in the **SafHashTable** constructor and may later be resized using the method **resize()**. **tableSize()** returns the current size of the hash-table.

Access

This is a pure virtual protected method. Must be implemented in sub-classes of **SafHashTable**.

Parameters

hObject Pointer to the hash key object.

Return Value

Must return an index within the hash-table calculated using the key *hObject*.

Example

```
uint32 VariableMap::hash( void* hObject ) const
{
   return (uint32)hObject % tableSize();
}
```

See Also: **SafHashTable::rehash**

SafHashTable::isEqual

virtual SafTBool isEqual(void* *hObject*, **void*** *object* **) const =0**

Description

This pure virtual method must check if the hash key object matches the object passed as argument. This method is called from other **SafHashTable** methods when looking up an object in the hash-table.

The class **SafHashTable** must be further sub-classed and this method must be overloaded**.**

Access

This is a pure virtual protected method. Must be implemented in sub-classes of **SafHashTable**.

Parameters

hObject Pointer to the hash key object.

object Pointer to the object.

Return Value

Must return *SafTTrue* if the hash key object matches the object, otherwise *SafTFalse*.

Example

```
SafTBool VariableMap::isEqual( void* hObject, void* object ) const
{
   return (int)hObject == ((VariableObject *)object)->id();
}
```

See Also: SafHashTable::doLookup, SafHashTable::doRemove

SafHashTable::itContains

`SafTBool itContains( void*` *`hObject`* `) const`

Description

This method checks if an object in the hash-table matches the supplied key.

Access

This function is protected. Can only be called from sub-classes of **SafHashTable**.

Parameters

hObject Pointer to the hash key object.

Return Value

Returns *SafTTrue* if the object is found in the hash-table, otherwise *SafTFalse*.

Example

```
SafTBool VariableMap::contains( int key ) const
{
    return itContains( (void *)key );
}
```

See Also: SafHashTable::doLookup

SafHashTable::rehash

`virtual uint32 rehash( void*` *`object`* `) const`

Description

This virtual method must return an index within the hash-table. The hash-table index must be calculated using information stored in the parameter *object*.

This method is called for all objects in the hash-table when calling the method **resize()**. It is not necessary to overload this method in sub-classes of **SafHashTable** if the method **resize()** is not used. The default implementation for this method only returns 0.

Use the method **tableSize()** to get the current size of the hash-table.

Access

This is a virtual protected method.

Parameters

object Pointer to the object to rehash.

Return Value

Must return an index within the hash-table calculated using information stored in the object.

Example

```
uint32 VariableMap::rehash( void* object ) const
{
   return ((VariableObject *)object)->id() % tableSize();
}
```

See Also: SafHashTable::resize, SafHashTable::tableSize

SafHashTable::removeAll

void removeAll(SafTBool *doDeleteElements* **= SafTFalse)**

Description

This method removes all the objects from the hash-table. If the parameter *doDeleteElements* is true the objects are deleted by invoking the method **doDelete()** for each element in the hash-table.

This method should be called from destructors of **SafHashTable** sub-classes to cleanup memory.

Access

This function is public.

Parameters

doDeleteElements *SafTTrue* if the objects should be deleted, otherwise *SafTFalse*.

See Also: SafHashTable::doDelete

SafHashTable::resize

SafTBool resize(uint32 *tableSize* **)**

Description

This method resizes the hash-table to the supplied table size. This size should reflect the number of objects put into or will be put into the hash-table.

For every object in the hash-table the method **rehash()** will be called to get new hash-table indexes. If this method is used the virtual method **rehash()** must be implemented in the sub-class of **SafHashTable**.

The size of the hash-table will work optimally when the table size is a prime number, for example 211, 509, 1021, 4091, 8191 or 16411.

Access

This function is public.

Parameters

tableSize New size of the hash-table.

Return Value

Returns *SafTTrue* if the hash-table was successfully resized, otherwise *SafTFalse*.

See Also: SafHashTable::SafHashTable, SafHashTable::rehash

SafHashTable::SafHashTable

```
SafHashTable( uint32 tableSize = 29 )
```

Description

This is the constructor for the class **SafHashTable**. It initialises the instance. The initial size of the hash-table is set to the supplied table size given in the parameter *tableSize*.

The size of the hash-table will work optimally when the table size is a prime number, for example 211, 509, 1021, 4091, 8191 or 16411.

The size of the hash-table may be resized at any time using the method **resize()**.

Access

This constructor is public.

Parameters

tableSize The initial size of the hash-table.

See Also: SafHashTable::~SafHashTable, SafHashTable::resize

SafHashTable::tableSize

```
uint32 tableSize() const
```

Description

This method returns the size of the hash-table.

Access

This function is public.

Return Value

Returns the size of the hash-table.

See Also: SafHashTable::SafHashTable, SafHashTable::resize

SafHashTable::~SafHashTable

```
virtual ~SafHashTable()
```

Description

This is the destructor for the class **SafHashTable**. It destroys the hash-table.

Note that the objects in the hash-table are not removed. In the sub-class destructor calling the method **removeAll()** will remove all the objects.

Access

This is a virtual public destructor.

Example

```
VariableMap::~VariableMap()
{
   removeAll( SafTTrue );
}
```

See Also: **SafHashTable::SafHashTable**

SafInterface

SafInterface is an abstract class used as a base class for the *SAF* classes **SafInternalInterface** and **SafExternalInterface**. **SafInterface** will only be used as a generic pointer to instances of sub-classes of **SafInternalInterface** and **SafExternalInterface**. Several methods in different *SAF* classes return a pointer or reference to a generic **SafInterface** instance.

Note that **SafInterface** must not in any circumstance be sub-classed directly in the application.

The **SafInterface** contains a set of virtual methods that normally will be overloaded in sub-classes. These methods are:

- **establish**
- **destroy**
- **onConnect**
- **onDisconnect**
- **onVariableUpdate**

As described above this class is used as a base class for the interface classes **SafExtenalInterface** and **SafInternalInterface**. Both internal and external interfaces must be further sub-classed.

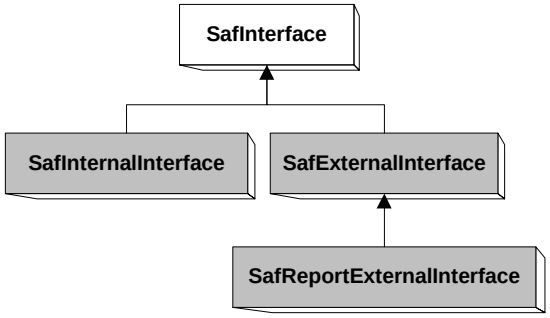
Internal interfaces must be developed with and used by a server to store information about its client and to access the functionality defined by the server within the clients. Internal interface objects use functionality implemented in the external interface objects of the client program.

External interfaces should hide *SWBus* functionality from the client module. An external interface should be developed by the same development team that created the server module because it access functionality only known in the server. An external interface is used by the client to access functionality offered by the server.

Between an internal interface in a server module and an external interface in a client module there is a logical connection. The physical connection is represented by a **SafConnection** sub-class instance. To better organize the application source code several logical connections can be represented by different **SafInterface** sub-classes. For instance one interface can be used to access server functionality for transferring data, another interface may be used to monitor the server. This is a simple example explaining why several interfaces can be useful.

SafInterface macros and functions are implemented to simplify the process of creating *SWBus* functions and to call interface object methods in the remote process.

A logical connection between an internal and external interface is not established before the **onConnect()** method are invoked in both the client and server module. The **establish()** method should create *SWBus* classes and functions, while the **onConnect()** method should get *SWBus STI*s of objects in the remote process.



Class Members

Constructors/Destructors

SafInterface	Constructs a **SafInterface** object.
~SafInterface	Destroys a **SafInterface** object.

Get Methods

connection	Returns the **SafConnection** sub-class instance associated with the interface.
interfaceClass	Returns the interface class.
interfaceId	Returns the unique interface identification.
interfaceType	Returns the interface type, *SafCExternal* or *SafCInternal*.
isConnected	Returns the state of the interface connection.
isEstablished	Returns true if the **establish()** method has been called.

Overriddable Methods

destroy	Is called to clean-up *SWBus* objects created in **establish()**.
establish	Is called to create *SWBus* classes and functions.
onConnect	Is called when the *logical* connection between an external and an internal interface instance are established.
onDisconnect	Is called when the *logical* connection is broken.
onVariableUpdate	Can be called when new variable updates are received.

SWBus Methods

call	Calls an *SWBus* function in the remote process.
declare	Declares an *SWBus* function.
sub	Sub-classes an *SWBus* class.

Initialisation Methods

initialize	Initialises the internal or external interface.

Macros

SafDInterfaceInfo	Declaration of struct used as input parameter to **call()**.
SafDDeclareInterface	Declares functionality used by the methods **establish()** and **destroy()**.
SafDDestroyInterface	Calls the method **destroy()**.
SafDImplementInterface	Implement functionality used by the methods **establish()** and **destroy()**.

Global Functions

SafFGetInterface	Returns a pointer to an interface object.

Member Functions

SafInterface::call

```
SbTSTI call( SbTSTI function,
             SbTSTI inParam,
             SbTSTI outParam )
```

Description

This function is used to call a **SafInterface** sub-class method in the remote process. It can not call a **SafInterface** sub-class method directly, but indirectly through an *SWBus* function. Enough information is stored in the input parameter *inParam* to access the correct connection object in the remote process. This extra information is added to the input parameter in this method.

The parameter *inParam* must be either *SbCNull* or must have been created using the method **sub()**. If the parameter *inParam* is *SbCNull* an internal *SWBus* parameter is created storing information about the interface object to call in the remote process. In the remote process using the global function **SafFGetInterface()** will get a pointer to the correct interface object. The input parameter passed to this global function is the same input argument that was passed to the *SWBus* function.

If the *inParam* is not *SbCNull* the *SWBus* class and the corresponding *C* struct must have the same layout. It is extremely important that the first field in the struct is the macro **SafDInterfaceInfo** or that it is derived directly from **SafInterfaceClassStruct**, otherwise unpredictable behaviour may be the result.

The *outParam* can be a predefined *SWBus* class or a class created using the *SWBus* function **SbSub()**.

When calling an interface method in the remote process always use the methods **declare()**, **sub()** and **call()**. Do not mix these functions with a direct call to **SbDeclare()**, **SbSub()** or **SbCall()**. It is extremely important to be aware of this because the function **SafFGetInterface()** can not be used to get correct interface object in the remote process if these functions are mixed. The exception is the output parameter. It is not necessary to use the method **sub()** to create the *SWBus* class because no extra information is used when returning from the *SWBus* function.

Note that even if a single variable of a predefined *SWBus* class should be passed as parameter to a remote function, it must be encapsulated in a user-defined class created using the method **sub()**.

Access

This is a protected method. Can only be called from sub-classes of **SafInterface**.

Parameters

function	The *SWBus STI* of the function to call in the remote process.
inParam	The *SWBus STI* of the input parameter object. Either *SbCNull* or an instance of a class created using sub().
outParam	The *SWBus STI* of the output parameter object. Pass *SbCNull* if no return value is needed.

Return Value

This method returns *SbCError* in the following situations:

- If the parameter *function* is not a valid *SWBus STI*.

- If the input parameter is *SbCNull* and it fails to create an internal *SWBus* object.
- If the type of the input parameter is not created using **sub()**.
- If the SWBus object's data area is NULL.

In all other situations the return value from **SbCall()** is returned.

Example

This is a comprehensive example demonstrating the use of internal and external interfaces. It demonstrates from an external interface how to call a function in the remote process and how to forward the function call to an internal interface.

The struct *SInParam* is used as input parameter to the *SWBus* function to call in the remote process.

```
struct SInParam
{
   SafDInterfaceInfo;
   int mValue;
};
```

The class *ClientEI* is a sub-class of **SafExternalInterface**. It is created in the client process. This method calls the *SWBus* function *sb_set()* in the server process where value is input argument to the internal interface method *ii_set()*. In this example *stiIn* and *stiFunc* are the *SWBus STI*s of the input class and the remote function, respectively. The *SWBus* function returns an integer.

```
int ClientEI::ei set( int value )
{
   SbTSTI inParam  = SbCreate( SbCPLocal, stiIn,    0, SbCNull, 0 );
   SbTSTI outParam = SbCreate( SbCPLocal, SbCInt32, 0, SbCNull, 0 );

   SInParam* in = (SInParam *)SbData( inParam );
   in->mValue = value;

   call( stiFunc, inParam, outParam );

   return *(int *)SbData( outParam );
}
```

In the server process the class *ServerII* is a sub-class of **SafInternalInterface**. When the external interface method *ei_set()* is called in the client process the method *ii_set()* will be called in *ServerII* class.

```
int ServerII::ii set( int value )
{
   return value * 10;
}
```

The *SWBus* function *sb_set()* is actually the function that is called when the *ClientEI* method *ei_set()* is called in the client task. The global function **SafDGetInterface()** is used to get the correct internal interface.

```
SbTSTI sb set( SbTSTI, SbTSTI, SbTSTI inParam,
                               SbTSTI outParam )
{
   ServerII* sI = (ServerII *)SafFGetInterface( inParam );

   SInParam* in  = (SInParam *)SbData( inParam );
   int*      out = (int *)SbData( outParam );

   *out = sI->ii_set( in->mValue );
   return outParam;
}
```

See Also: **SafInterface::sub, SafInterface::declare, SafDGetInterface, SafDInterfaceInfo, SafInternalInterface, SafExternalInterface, SbSub, SbCall, SbDeclare**

SafInterface::connection

```
SafConnection* connection() const
```

Description

This method returns a pointer to an instance of a sub-class of **SafConnection** that is associated with the interface.

Access

This function is public.

Return Value

Returns a pointer to a connection object.

See Also: SafConnection

SafInterface::declare

```
SbTSTI declare( const char* name,
                SbTSTI  inputClass,
                SbTSTI  outputClass,
                SbTCode function )
```

Description

This method declares a function class called *name*. Instances of this class are callable functions that can be used as arguments to the method **call()**. *inputClass* and *outputClass* specify the classes of the formal input and output parameters respectively. The actual function code is situated at the address *function*.

If the parameter *inputClass* is set to *SbCNull* an internal *SWBus* class is declared as input to the *C* function. In that case the input parameter to the method **call()** must also be called with *SbCNull* .

When declaring an *SWBus* function **declare()** should be used. Do not mix the methods **declare()**, **sub()** and **call()** with a direct call to **SbDeclare()**, **SbSub()** or **SbCall()**. It is extremely important to be aware of this because the function **SafFGetInterface()** can not be used to get correct interface object in the remote process if these functions are mixed. The exception is the output parameter. It is not necessary to use the method **sub()** to create the *SWBus* class because no extra information is used when returning from the *SWBus* function.

Access

This is a static public function.

Parameters

name	Name of the *SWBus* function class.
inputClass	*SWBus* class of the input parameter.
outputClass	*SWBus* class of the output parameter.
function	The actual address of the function.

Return Value

Returns the *SWBus STI* of the new function class. *SbCError* is returned if the operation failed.

Example

This example demonstrates the use of **declare()** to create an *SWBus* function class. The class *IIClass* is a sub-class of **SafInternalInterface**. The *C* function *sb_func()* accepts an input argument of type *mInputArgStruct* and an output argument of type *SbCCString*.

```
SbTSTI sb func( SbTSTI o, SbTSTI f, SbTSTI in, SbTSTI out )
{
  /* Do something with the in and out parameter */

   return out;
}


void IIClass::create()
{
   mInputArgClass = sub( "mInputArgStruct" );
   /* Skipped code for adding fields here ... */

   mDescF = declare( 0, mInputArgClass, SbCCString, sb func );
}
```

See Also: SafInterface::call, SafInterface::sub

SafInterface::destroy

`virtual SafStatus destroy()`

Description

This virtual method is called when a interface object is deleted and the macro **SafDDestroyInterface()** is put into the destructor of the **SafInterface** sub-class.

Normally *SWBus* classes and functions created in the method **establish()** should be deleted in this method.

It is possible to have this method called only when deleting the last interface object of a specific sub-class of **SafInterface**. To get this functionality use the macros **SafDDeclareInterface()** and **SafDImplementInterface()**. For more information about these macros refer to the macro section. When using these macros the method **establish()** will only be called when creating the first instance of a sub-class of **SafInterface**.

Access

This is a virtual protected method.

Return Value

Should return *Saf_OK* on success, otherwise an error code.

Example

This example demonstrates the use of the macros **SafDDeclareInterface()**, **SafDImplementInterface()** and **SafDDestroyInterface()**. In the method **establish()** *SWBus* objects are created. This method is called only once for the class *IIClass*. When the last instance of the class *IIClass* is deleted the method **destroy()** will be called. This method should destroy *SWBus* objects created in the method **establish()**.

```
class IIClass : public SafInternalInterface
{
   …
   ~IIClass();
   SafStatus establish();
   SafStatus destroy();
   SafDDeclareInterface;
};

SafDImplementInterface( IIClass );

IIClass::~IIClass()
{
   …
   SafDDestroyInterface;
}

SafStatus IIClass::establish()
{
   /* Create SWBus classes and functions */
   return Saf OK;
}

SafStatus IIClass::destroy()
{
   /* Destroy SWBus classes and functions created in establish */
   return Saf OK;
}
```

See Also: **SafInterface::establish, SafDDeclareInterface, SafDImplementInterface, SafDDestroyInterface**

SafInterface::establish

`virtual SafStatus establish()`

Description

This virtual method is automatically invoked by *SAF* when the member method **initialize()** is called. It may be overloaded in sub-classes of **SafInterface**. By default **establish()** will be called each time **initialize()** is called for instances of a specific **SafInterface** sub-class.

To prevent that the method **establish()** is invoked each time **initialize()** is called for a new interface instance the following macros can be used, **SafDDeclareInterface** and **SafDImplementInterface**. If these macros are used the **establish()** method will be invoked only once for each specific sub-class of **SafInterface**.

Normally this method should declare *SWBus* classes and functions relevant for the **SafInterface** sub-class.

Note that on the time when this method is invoked by *SAF* no *logical* connection between the external and the internal interface exist. Do not get any *SWBus STI*s of objects in the remote process in this method. A *logical* connection is not established before the **onConnect()** method is called. It is guaranteed that the **establish()** method is invoked for the external and the internal interface before the **onConnect()** method is called.

Access

This is a virtual protected method.

Return Value

Should return *Saf_OK* on success, otherwise an error code.

Example

For a complete example using macros see the description for the method **destroy()**. Note that the example also uses the macro **SafDDestroyInterface** to call the **destroy()** method. This macro can be omitted if the **destroy()** method is not needed.

The following example demonstrated a typical **establish()** method that declares and creates *SWBus* variables and functions.

```
SafStatus IIClass::establish()
{
   SbTSTI cl = sub( "MyClass" );
   SbAdd( cl, SbCCInt32, "state", NULL );
   SbAdd( cl, SbCCFloat, "level", NULL );

   SbTSTI fd = declare( NULL, cl, SbCNull, sb func );
   SbTSTI fc = SbCreate( SbCPLocal, fd, "F1", SbCNull, NULL );

   return Saf OK;
}
```

See Also: **SafInterface::destroy, SafInterface::onConnect, SafDDeclareInterface, SafDImplementInterface, SafDDestroyInterface**

SafInterface::initialize

`virtual SafStatus initialize() =0`

Description

This is a pure virtual function implemented in the **SafInterface** sub-classes **SafExternalInterface** and **SafInternalInterface**. It initialises the **SafInterface** sub-class instance. The implementation depends on whether the class inherits from **SafInternalInterface** or **SafExternalInterface**.

For a detailed description refer to **SafExternalInterface::initialize()** and **SafInternalInterface::initialize()**.

Access

This is a pure virtual public method.

Return Value

Returns *Saf_OK* on success, otherwise an error code.

See Also: **SafInternalInterface::intitialize, SafExternalInterface::initialize**

SafInterface::interfaceClass

`const char* interfaceClass() const`

Description

Returns the interface class for this **SafInterface** sub-class.

Access

This function is public.

Return Value

Returns the interface class name.

Example

This example checks if the generic interface pointer passed as argument is of type MONITOR.

```
SafTBool isMonitor( SafInterface* itf )
{
   return strcmp( itf->interfaceClass(), "MONITOR" ) ?
                                            SafTTrue :
                                            SafTFalse;
}
```

See Also: SafInterface::SafInterface

SafInterface::interfaceId

`int32 interfaceId() const`

Description

This method returns the unique interface identification for this object. The interface identification is generated in the **SafExternalInterface** constructor. When creating an internal interface the *interfaceId* passed as parameter to the **SafConnection::createInterface** must be passed to the **SafInterface** constructor without modifications.

Access

This function is public.

Return Value

Returns the unique interface identification.

See Also: SafConnection::createInterface

SafInterface::interfaceType

`virtual int32 interfaceType() const =0`

Description

This is a pure virtual method returning the type of interface. This method is overloaded in the **SafInterface** sub-classes **SafExternalInterface** and **SafInternalInterface**.

Access

This is a pure virtual public method. It is implemented in the sub-classes **SafExternalInterface** and **SafInternalInterface**.

Return Value

Returns *SafCInternal* or *SafCExternal* depending on the interface type.

See Also: **SafExternalInterface::interfaceType,**
SafInternalInterface::interfaceType

SafInterface::isConnected

```
SafTBool isConnected() const
```

Description

This method checks if the connection to the external task is established.

Access

This function is public.

Return Value

Returns *SafTTrue* if the connection is open, otherwise *SafTFalse*.

See Also: **SafConnection::isConnected**

SafInterface::isEstablished

```
SafTBool isEstablished() const
```

Description

This method checks whether the **establish()** method has been called or not.

Access

This function is public.

Return Value

Returns *SafTTrue* if the **establish()** method has been called, otherwise *SafTFalse*.

See Also: **SafInterface::establish**

SafInterface::onConnect

```
virtual SafStatus onConnect()
```

Description

This virtual method is called by *SAF* when a logical connection between an external interface and an internal interface are established. Normally this method is overloaded in sub-classes of **SafExternalInterface** and **SafInternalInterface**. The default implementation returns *Saf_OK*.

The **onConnect()** method is called in both **SafExternalInterface** and **SafInternalInterface** sub-class instances.

Normally this method should get *SWBus STI*s of objects in the remote process. Do not create *SWBus* classes or functions in this method. Such functionality should be put into the method **establish()**.

Access

This is a virtual protected method.

Return Value

Should return *Saf_OK* on success, otherwise an error code.

Example

This example demonstrates how to implement a typical **onConnect()** method. It gets *SWBus STI*s of objects in the remote process. The class *EIClass* is a sub-class of **SafExternalInterface**.

```
SafStatus EIClass::onConnect()
{
   static char* sNames[] =
   {
      "myFunc",
      "var1",
      "var2",
      NULL
   };

   SbTSTI rSti = connection()->remoteSTI();
   SbTSTI lSti = SbIds( rSti, sNames );

   mFuncId        = SbHead( lSti ); lSti = SbTail( lSti );
   SbTSTI rVar1Id = SbHead( lSti ); lSti = SbTail( lSti );
   SbTSTI rVar2Id = SbHead( lSti );

   // Link var1 and var2 to local variable
   SbTSTI lVar1Id = SbId( SbCPLocal, "var1" );
   SbTSTI lVar2Id = SbId( SbCPLocal, "var2" );

   mVar1Id = SbLink( lVar1Id, rVar1Id, SbCBPNull );
   mVar2Id = SbLink( lVar2Id, rVar2Id, SbCBPNull );

   return Saf OK;
}
```

See Also: **SafInterface::establish, SafInterface::onDisconnect, SbIds, SbId, SbHead, SbTail, SbLink**

SafInterface::onDisconnect

`virtual SafStatus onDisconnect()`

Description

This is a virtual method that is automatically called by *SAF* when the connection to the external task is broken or gracefully disconnected. It will also be called when the connected **SafInterface** sub-class instance is deleted in the external task.

The method may be overloaded in sub-classes of **SafInterface**. A typical implementation will clear remote *SWBus STI*s.

Access

This is a virtual protected method.

Return Value

Should return *Saf_OK* on success, otherwise an error code.

Example

This example demonstrates how a typical **onDisconnect()** method will be implemented. The class *EIClass* is derived from **SafExternalInterface**. *mVar1Id*, *mVar2Id* and *mFunc* are attributes in the class *EIClass*.

```
SafStatus EIClass::onDisconnect()
{
   mVar1Id = SbCNull;
   mVar2Id = SbCNull;
   mFunc   = SbCNull;

   return Saf OK;
}
```

See Also: SafInterface::onConnect

SafInterface::onVariableUpdate

```
virtual SafStatus onVariableUpdate( int32   numVariables,
                                    SbTSTI* variables )
```

Description

This method should be called when updated variables should be forwarded directly to a **SafInterface** sub-class instance. The default implementation only returns *Saf_OK*. Note that this method is not automatically invoked by *SAF*.

This function may be overloaded in the **SafExternalInterface** or **SafInternalInterface** sub-class. The parameter *numVariables* is the number of variables updated. *variables* is an array of *SWBus STI* variables.

Access

This is a virtual public method.

Parameters

numVariables	Number of variables updated.
variables	Array of updated variables.

Return Value

Should return *Saf_OK* on success, otherwise an error code.

Example

This example demonstrates the use of *TheTask::onVariableUpdate()* and how it forwards the function call to the **SafInterface** sub-class *IIUpdate::onVariableUpdate()*. In this example the attribute *mUpdate* is of class *IIUpdate,* which is a sub-class of **SafInternalInterface**. The class *TheTask* is a sub-class of **SafTask**.

```
SafStatus TheTask::onVariableUpdate( const char* taskName,
                                     int32        numVars,
                                     SbTSTI*      vars )
{
   /* This method is automatically called by SAF */
   return mUpdate->onVariableUpdate( numVars, vars );
}


SafStatus IIUpdate::onVariableUpdate( int32   numVars,
                                      SbTSTI* vars )
{
   for ( int j = 0; j < numVars; j++ )
      update( vars[j] );

   return Saf OK;
}
```

See Also: **SafTask::onVariableUpdate, SafConnection::onVariableUpdate**

SafInterface::SafInterface

```
SafInterface( SafConnection* connection,
              const char*    interfaceClass,
              int32          interfaceId )
```

Description

This method is the constructor for the class **SafInterface**. It initialises the **SafInterface** instance. The parameter *connection* is a pointer to a **SafConnection** sub-class instance that the interface is associated with. The **SafInterface** instance is registered with the **SafConnection** object.

The *interfaceClass* is the interface class name. It is extremely important that different interface classes have different interface class names. The *interfaceId* is a unique identifier that is generated in the **SafExternalInterface** constructor. When creating **SafInternalInterface** sub-class instances this *interfaceId* is passed as an argument to the **SafConnection::createInterface()** method.

Access

This constructor is public.

Parameters

connection	Pointer to a **SafConnection** sub-class instance.
interfaceClass	Name of the interface class.
interfaceId	Unique interface identifier generated in the **SafExternalInterface** constructor.

See Also: **SafInterface::~SafInterface, SafExternalInterface::SafExternalInterface, SafInternalInterface::SafInternalInterface, SafConnection::createInterface, SafConnection::destroyInterface**

SafInterface::sub

static SbTSTI sub(const char* *name* **)**

Description

This static method creates a new *SWBus* class called *name*. The new class will be a sub-class of an internal *SAF* class. Instances of classes created using **sub()** should be used as input parameter to the method **call()**. Use **SbAdd()** to add fields to this class.

The *SWBus* class created using **sub()** and the corresponding *C* struct must have the same layout. It is extremely important that the first field in the *C* struct is the macro **SafDInterfaceInfo** or that it is derived directly from **SafInterfaceClassStruct**.

When calling an interface object method in the remote process always use the methods **declare()**, **sub()** and **call()**. Do not mix these functions with a direct call to **SbDeclare()**, **SbSub()** or **SbCall()**. It is extremely important to be aware of this because the function **SafFGetInterface ()** can not be used to get correct interface object in the remote process if these functions are mixed. The exception is the output parameter. It is not necessary to use the method **sub()** to create the *SWBus* class because no extra information is used when returning from the *SWBus* function.

Access

This is a static public function.

Parameters

name Name of the new *SWBus* class.

Return Value

Returns the *SWBus STI* of the new class, or *SbCError* if the operation failed.

Example

This example demonstrates how to create an *SWBus* class and the layout of the corresponding *C* struct.

```
struct ValveIn
{
  SafDInterfaceInfo;
  float mValue;
  int   mState;
}

void IIClass::create()
{
   SbTSTI c = sub( "ValveIn" );
   SbAdd( c, SbCFloat, "mValue", NULL );
   SbAdd( c, SbCInt32, "mState", NULL );
}
```

See Also: SafInterface::declare, SafInterface::call, SbSub, SbAdd

SafInterface::~SafInterface

virtual ~SafInterface()

Description

This method is the destructor for the class **SafInterface**. It frees memory allocated by the **SafInterface** object and removes itself from the associated connection instance. Its functionality is inherited and expanded by both **SafExternalInterface** and **SafInternalInterface**

Access

This is a virtual public destructor.

See Also: **SafInterface::SafInterface**

Macros

SafDInterfaceInfo

`SafDInterfaceInfo`

Description

This macro is used when creating a *C* struct corresponding to an *SWBus* class created using the **SafInterface** method **sub()**. It is very important that this macro is put first in the struct definition otherwise undefined behaviour may be the result.

It is also possible to define a struct deriving directly from **SafInterfaceClassStruct**.

Example

```
struct sIn1
{
   SafInterfaceInfo;
   …           /* Add fields */
};


struct sIn2 : public SafInterfaceClassStruct
{
   …            /* Add fields */
};
```

See Also: **SafInterface::sub, SafInterface::call**

SafDDeclareInterface

`SafDDeclareInterface`

Description

This macro together with **SafDImplementInterface** is used to prevent that the **establish()** method is called more than once for a specific **SafInterface** sub-class. When these macros are used only the first instance created of a sub-class of **SafInterface** will call the **establish()** method.

This macro must be put inside the class definition. Normally this class is defined in a header file.

Note that **SafDImplementInterface** must be added outside the class definition if this macro is used. If not the compiler will produce several error messages as a result.

Example

This is an example that defines an interface class using the class **SafInternalInterface** as a base class. It adds the **SafDDeclareInterface** macro. This is a typical implementation of a **SafInterface** sub-class. It is defined in a header file.

```
class IInterface: public SafInternalInterface
{
 private:
    …
    SafStatus establish();

 public:
    IInterface( SafConnection* c, int interfaceId );
    ~IInterface ();

    …

 SafDDeclareInterface;
};
```

The implementation continues below in the *C* file. The macro **SafDImplementInterface()** is used to define the static variables declared in the macro **SafDDeclareInterface**. Note the use of the class name as argument.

```
SafDImplmentInterface( IInterface );

IInterface::IInterface( SafConnection* c, int interfaceId ) :
   SafInternalInterface( c, "IICLASS", interfaceId )
{
  …
}

IInterface::~IInterface()
{
   …
}

SafStatus IInterface::establish()
{
  …

  return Saf OK;
}
```

See Also: SafInterface::establish, SafDImplementInterface

SafDDestroyInterface

`SafDDestroyInterface`

Description

This macro is used to call the virtual method **destroy()** from the destructor of a **SafInterface** sub-class. Put this macro into the destructor of the sub-class.

If using the macros **SafDDeclareInterface** and **SafDImplementInterface** the method **destroy()** will only be called when deleting the last instance of a **SafInterface** sub-class. If these macros are not used it will be called each time an instance is deleted.

Example

```
IInterface::~IInterface()
{
   …
   SafDDestroyInterface;
}

SafStatus IInterface::destroy()
{
   …

   return Saf OK;
}
```

See Also: **SafInterface::~SafInterface, SafInterface::destroy, SafDDeclareInterface, SafDImplementInterface**

SafDImplementInterface

`SafDImplementInterface(` *SAFCLASS* **`)`**

Description

This macro is used together with the macro **SafDDeclareInterface** to prevent that the **SafInterface** sub-class method **establish()** is called more than one time for a given class. For more information refer to **SafDDeclareInterface**.

Parameters

SAFCLASS *Name of a **SafInterface** sub-class.*

Example

See example for **SafDDeclareInterface**.

See Also: **SafDeclareInterface**

Global Functions

SafFGetInterface

`SafInterface* SafFGetInterface ( SbTSTI` *in* **`)`**

Description

This function returns a pointer to a **SafInterface** instance based on the information passed in the parameter *in*. The parameter *in* must be created in the remote process using the **SafInterface** method **sub()**.

Calling the **SafInterface** method **call()** without an input argument still creates enough information for **SafFGetInterface()** to get proper **SafInterface** instance. Refer to **sub()** and **call()** for a detailed description.

This function should only be called from an *SWBus* function that was invoked from a remote process using the **SafInterface** method **call()**.

Parameters

in Input parameter passed to the *SWBus* function.

Return Value

Returns a pointer to a **SafInterface** object if it was found in *SAF*, otherwise *NULL*.

Example

This example demonstrates a typical implementation of an *SWBus* function that calls a method in a **SafInterface** object.

```
SbTSTI subscribeCB( SbTSTI, SbTSTI, SbTSTI in, SbTSTI out )
{
    IInterface* ii = (IInterface *)SafFGetInterface( in );
    if ( ii )
       ii->subscribe( in );
    else
       cout << "Failed to find interface object" << endl;

    return out;
}
```

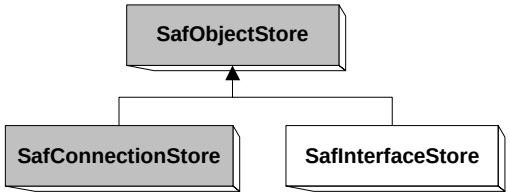
See Also: SafInterface::sub, SafInterface::call, SafFGetConnection

SafInterfaceStore

A **SafInterfaceStore** object maintains an ordered collection of any object derived from the class **SafInterface**. The objects are put into a double linked list.

SafInterface objects can be added and removed from the list. The list can be traversed.

This class may be sub-classed further to get extended functionality present in the generic base class **SafObjectStore**.



Class Members

Construction/Destruction	
~SafInterfaceStore	Destroys **SafInterfaceStore** object.

Operations	
append	Appends a new interface to the list.
doDelete	Deletes an interface object.
remove	Removes an interface from the list.

Get Methods	
contains	Checks if an interface is in the list.
getNext	Used to get the next interface in the list.
isEqual	Checks if two interfaces are equal.

Member Functions

SafInterfaceStore::append

```
SafTBool append( const SafInterface* itf )
```

Description

Appends the **SafInterface** sub-class object to the end of the list.

Access

This function is public.

Parameters

itf **SafInterface** sub-class instance to add to the list.

Return Value

Returns *SafTTrue* if the interface is successfully appended to the list. If the operation failed *SafTFalse* is returned.

See Also: SafInterfaceStore::remove

SafInterfaceStore::contains

```
SafInterface* contains( int32 interfaceId ) const
```

Description

Traverses the list and returns the **SafInterface** sub-class instance with the interface identification equal to the parameter *interfaceId*. If the *interfaceId* does not exist in the list NULL is returned.

Access

This function is public.

Parameters

interfaceId The object's *interfaceId* to look for in the list.

Return Value

Returns a pointer to the **SafInterface** sub-class if the object is found, otherwise NULL.

See Also: SafInterfaceStore::append, SafInterfaceStore::remove

SafInterfaceStore::doDelete

```
void doDelete( void* object )
```

Description

This protected method deletes the **SafInterface** sub-class instance. The parameter *object* points to the instance to delete. The instance is deleted using the *C++* operator *delete*.

This method is called within the **SafInterfaceStore** and **SafObjectStore** methods. Note that this method is only of interest to developers who sub-classes **SafInterfaceStore**.

Access

This is a virtual protected method.

Parameters

object The instance to delete.

See Also: SafInterfaceStore::remove, SafInterfaceStore::isEqual

SafInterfaceStore::getNext

SafInterface* getNext(SafPOSITION& *pos*) const

Description

This method is used when traversing the **SafInterface** sub-class objects in the list. It returns a pointer to current interface object. The parameter *pos* is set to point to the next interface object in the list.

To get the first object in the list the parameter *pos* must be initialised by calling the base class method **getHeadPosition()**.

Access

This function is public.

Parameters

pos	A reference to the position of the interface object to return. After the function call the position is pointing to the next interface. When reaching the end of the list the value of this variable is NULL. Use NULL as a termination condition in a loop.

Return Value

Returns a pointer to the **SafInterface** sub-class object.

Example

This example demonstrates how to traverse a list of **SafInterface** objects. The class *TheConnection* is derived from **SafConnection**. The **SafConnection** method **externalInterfaceStore()** is used to get a reference to the external interfaces associated with the connection object.

```
void TheConnection::printInterfaceClasses()
{
   SafInterfaceStore& list = externalInterfaceStore();
   SafPOSITION        pos  = list.getHeadPosition();

   while ( pos )
   {
      SafInterface* itf = list.getNext( pos );

      cout << "Interface class " << itf->interfaceClass()
           << endl;
   }
}
```

See Also: **SafConnection**

SafInterfaceStore::isEqual

SafTBool isEqual(const void* *o1*, const void* *o2*) const

Description

Checks if the objects *o1* and *o2* are equal.

This method is called within **SafInterfaceStore** and **SafObjectStore** methods. Note that this method is only of interest to developers who sub-classes **SafInterfaceStore**.

Access

This is a virtual protected method.

Parameters

o1	Pointer to interface object 1.
o2	Pointer to interface object 2.

Return Value

Returns *SafTTrue* if the objects are equal, otherwise *SafTFalse*.

See Also: SafInterfaceStore::doDelete

SafInterfaceStore::remove

```
SafTBool remove( const SafInterface* itf,
                 SafTBool deleteElement = SafTTrue )

SafTBool remove( int32    interfaceId,
                 SafTBool deleteElement = SafTTrue )
```

Description

This method removes the **SafInterface** sub-class instance given as interface pointer or interface identification from the list. If the parameter *deleteElement* is true the interface object is deleted.

Access

These functions are public.

Parameters

itf	Pointer to the interface object to remove from the list.
interfaceId	Interface identification of the object to remove from the list.
deleteElement	The interface object is deleted if set to *SafTTrue*. The object will not be deleted if set to *SafTFalse*.

Return Value

Returns *SafTTrue* if the object is successfully removed from the list, or *SafTFalse* if the operation failed.

See Also: SafInterfaceStore::append

SafInterfaceStore::~SafInterfaceStore

```
~SafInterfaceStore()
```

Description

This method is the destructor for the class **SafInterfaceStore**. It removes all the objects from the list. Note that the objects in the list are not deleted.

Access

This destructor is public.

Example

This example demonstrates how to delete all the objects in the list. The base class method **removeAll()** is used to delete the interface objects.

```
void remove( SafInterfaceStore* itfStore )
{
   // Delete all interface objects
   itfStore->removeAll( SafTTrue );

   delete itfStore;
}
```

See Also: **SafObjectStore::removeAll**

SafInternalInterface

A **SafInternalInterface** sub-class instance is a server's interface to a client program. It stores information about its client and is used to access functionality defined by a server within a client. Internal interface objects use functionality implemented in an external interface objects of a client program.

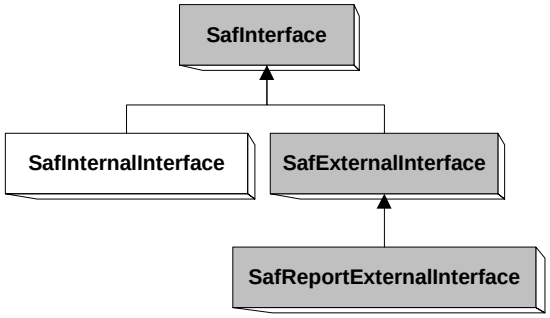
SafInternalInterface must be further sub-classed and functionality specific for the interface must be added. A server program may have several interfaces. These interfaces can be instances of the same class or from completely different classes with different functionality. **SafInternalInterface** is derived from **SafInterface.**

SAF offers macros to simplify the process of creating several interfaces of a class. Refer to the **SafInterface** macro section for further details.

When an external interface in the client process is initialised, a message is sent to the server process. *SAF* will automatically call the virtual **SafConnection** sub-class method **createInterface()**. It is extremely important that the information passed to **createInterface()** are used when creating the internal interface. The parameter *interfaceId* must be passed directly to the **SafInternalInterface** constructor without modifications. The character string *interfaceClass* is used to distinguish between the internal interfaces a server can create. A server program may support several **SafInternalInterface** sub-classes with different functionality. Depending on the parameter *interfaceClass* different instances of **SafInternalInterface** sub-classes may be created.

It is important that the internal interface is initialised using the method **initialize()** after it has been created. If not, the logical connection between the objects will not be established. Note that the external interface in the server process and the internal interface in the client process do not have a logical connection before the virtual method **onConnect()** is called.

For a detailed description of the base class refer to **SafInterface**.



Class Members

Construction/Destruction

SafInternalInterface	Constructs **SafInternalInterface** object.
~SafInternalInterface	Destroys **SafInternalInterface** object.

Overloaded Methods

interfaceType	Returns *SafCInternal*.
initialize	Initialises the internal interface.

Member Functions

SafInternalInterface::interfaceType

```
int interfaceType() const
```

Description

Returns the interface type. This method is declared as pure virtual in the base class **SafInterface**.

Access

This is a public method.

Return Value

Returns *SafCInternal*

See Also: **SafExternalInterface::interfaceType, SafInterface::interfaceType**

SafInternalInterface::initialize

```
SafStatus initialize()
```

Description

This method initialises the **SafInternalInterface** object. It calls the **establish()** method and finally calls the **onConnect()** method.

Note that this method must be called after the internal interface is created. A good practice is to call this method in **SafConnection::createInterface()**.

Access

This function is public.

Return Value

If the method **establish()** fails the error code from the function is returned, otherwise the return value from the method **onConnect()** is returned.

Example

This example demonstrates a typical **SafConnection::createInterface()** method creating two different interfaces depending on the *interfaceClass*. The class *Connection* is a subclass of **SafConnection**. The classes *DataDis* and *Monitor* are derived from **SafInternalInterface**.

```
SafStatus Connection::createInterface( const char* interfaceClass,
                                       int         interfaceId )
{
   SafInterface* itf;

   if ( !strcmp( interfaceClass, "DATADIS" ) )
      itf = new DataDis( this,
                         interfaceClass, interfaceId );
   else
   if ( !strcmp( interfaceClass, "MONITOR" ) )
      itf = new Monitor( this,
                         interfaceClass, interfaceId );
   else
      return Saf Error;

   itr->initialize();  // Intitialise internal interface

   return Saf OK;
}
```

See Also: **SafInterface::establish, SafInterface::onConnect, SafConnection::createInterface**

SafInternalInterface::SafInternalInterface

```
SafInternalInterface( SafConnection* connection,
                      const char*    interfaceClass,
                      int            interfaceId )
```

Description

This is the constructor for the class **SafInternalInterface**. It initialises the object and registers itself with the associated **SafConnection** instance.

Access

This constructor is public.

Parameters

connection	Pointer to **SafConnection** sub-class instance.
interfaceClass	The name of the interface class.
interfaceId	A unique interface identifier created in the external interface in the remote process.

Example

The following example demonstrates a typical constructor for the class *IIClass* and initialisation of the class **SafInternalInterface**.

```
IIClass::IIClass( SafConnection* c, const char* itfClass,
                  int itfId ) :
   SafInternalInterface( c, itfClass, itfId )
{

   /* Initialisation of member attributes */

}
```

See Also: **SafInternalInterface::~SafInternalInterface**

SafInternalInterface::~SafInternalInterface

```
~SafInternalInterface()
```

Description

This is the destructor for the class **SafInternalInterface**. It removes the interface object from the associated connection object's interface list.

Access

This destructor is public.

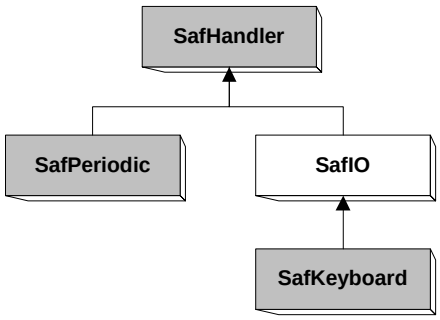
See Also: **SafInternalInterface::SafInternalInterface**

SafIO

The class **SafIO** is used when listening for input or output on file descriptors or sockets.

This class must be sub-classed and the method **onAction()** must be overloaded in the sub-class. The **onAction()** method will be called as a response to activity on an associated file descriptor or socket.

The class **SafIO** is derived from the base class **SafHandler**. For a complete description of the base class member methods refer to the **SafHandler** section.



Class Members

Construction/Destruction	
SafIO	Constructs **SafIO** object.
~**SafIO**	Destroys **SafIO** object.

Operations	
addSocket	Adds a new socket to the file descriptor set.
changeFds	Replaces the existing file descriptor set.
removeSocket	Removes a socket from the file descriptor set.

Member Functions

SafIO::addSocket

```
SafStatus addSocket( SOCKET s )
```

Description

Adds the socket *s* passed as argument to the current file descriptor set. Sockets can only be added if the object is not in a running state.

Access

This function is public.

Parameters

s	The socket to add to the current file descriptor set.

Return Value

Returns *Saf_OK* if the socket was added successfully to the file descriptor set, or *Saf_SbIsRunning* if the operation failed.

See Also: **SafIO::SafIO, SafIO::removeSocket, SafIO::changeFds, SafHandler::suspend, SafHandler::restart**

SafIO::changeFds

`SafStatus changeFds( fd_set&` *`newFds`* **`)`**

Description

This method replaces the existing file descriptor set with the new file descriptor set passed as argument. This operation only succeeds if the object is not in a running state.

Access

This function is public.

Parameters

newFds The new file descriptor set.

Return Value

Returns *Saf_OK* if the file descriptor set passed as argument where successfully replacing the existing file descriptor set, or *Saf_SbIsRunning* if the operation failed.

See Also: **SafIO::SafIO, SafIO::addSocket, SafIO::removeSocket, SafHandler::suspend, SafHandler::restart**

SafIO::removeSocket

`SafStatus removeSocket( SOCKET` *`s`* **`)`**

Description

Removes the socket *s* passed as argument from the current file descriptor set. Sockets can only be removed if the object is not in a running state.

Access

This function is public.

Parameters

s The socket to remove from the current file descriptor set.

Return Value

Returns *Saf_OK* if the socket was removed successfully from the file descriptor set, or *Saf_SbIsRunning* if the operation failed.

See Also: **SafIO::addSocket, SafIO::changeFds, SafIO::suspend, SafIO::restart**

SafIO::SafIO

SafIO()

SafIO(fd_set& *fds* **)**

Description

These methods are constructors for the class **SafIO**.

The constructor **SafIO()** initialises the object and clears current file descriptor set. Use the methods **addSocket()** or **changeFds()** to modify the object's file descriptor set.

The constructor **SafIO(fd_set&** *fds* **)** initialises the object and sets current file descriptor set equal to the argument *fds*.

Use the base class member method **activate()** to activate the IO handler.

Access

These constructors are public.

Parameters

fds	The new file descriptor set.

Example

This comprehensive example demonstrates the use of the class *AudioIO* for reading input data when there is activation on a SOCKET. It is an audio example. The class *AudioIO* reads audio data from a sound card when the sound is higher than a threshold. The message is sent from the audio card software driver using TCP/IP. The audio library functions (prefixed Au in this example) are not essential in this example and are therefore skipped.

```
#include <saf/saf.h>



class AudioIO : public SafIO
{
 private:
   SafStatus onAction();
   …

 public:
   AudioIO();
   ~AudioIO();
};


AudioIO::AudioIO() : SafIO()
{
  // Open the audio library. It returns a SOCKET.
   SOCKET s = AuOpen();
   addSocket( s );

   activate();   // Activate the SWBus IO object.
}

AudioIO::~AudioIO()
{
   AuClose();    // Close the audio library.
}

SafStatus AudioIO::onAction()
{
   AuBuffer buffer;
   AuRead( buffer );   // Read audio data from the SOCKET.

   update( buffer );   // Do something with the audio data.

   return Saf OK;
}
```

See Also: **SafIO::~SafIO, SafIO::addSocket, SafIO::changeFds, SafIO::removeSocket, SafHandler::activate**

SafIO::~SafIO

~SafIO()

Description

This method is the destructor for the class **SafIO**. It deletes the *SWBus* object.

Access

This destructor is public.

See Also: **SafIO::SafIO**

SafKeyboard

The class **SafKeyboard** is used when listening for input from the keyboard.

This method must be further sub-classed and the following virtual methods may be overloaded:

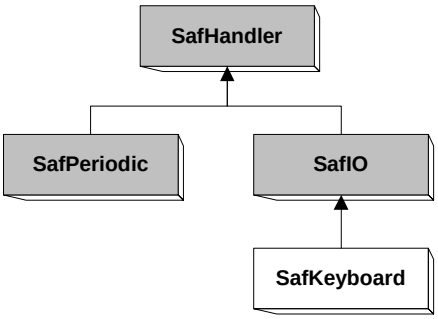
- **onKeyboardInputLine**
- **onKeyboardInputChar**

The method **onKeyboardInputChar()** is activated on single keystrokes (depends on the mode of the terminal window), while **onKeyboardInputLine()** is activated when the return key is pressed.

The virtual methods **onAction()**, **onSuspend()**, **onRestart()** and **onActivate()** declared in the base class **SafHandler** are not available in **SafKeyboard** sub-classes. These virtual methods are declared as private in the class **SafKeyboard**.

Note that only one instance of a keyboard handler can be active simultaneously.

On *Windows* the keyboard handler is only available when the program is compiled with multithreaded options.



Class Methods

Construction/Destruction	
SafKeyboard	Constructs **SafKeyboard** object.
~SafKeyboard	Destroys **SafKeyboard** object.

Overriddable Methods	
onKeyboardInputChar	Reads one character from the keyboard.
onKeyboardInputLine	Reads a line from the keyboard.

Member Functions

SafKeyboard::onKeyboardInputChar

virtual SafStatus onKeyboardInputChar(char *c* **)**

Description

This virtual method is activated when a key is pressed on the keyboard. The parameter *c* is the key being pressed.

This function may be overloaded in sub-classes of **SafKeyboard**. Normally either **onKeyboardInputChar()** or **onKeyboardInputLine()** is overloaded, not both.

Note that the behaviour of this method depends on the terminal window the program is running in. Some terminal windows buffer keyboard strokes until the return key is pressed and sends the input as a single line. The **onKeyboardInputChar()** will when running in such environments, not be activated before the return key is pressed.

Access

This is a virtual protected method.

Parameters

c	The character pressed on the keyboard.

Return Value

Must returns *Saf_OK* on success, or an error code if the operation failed.

Example

This example demonstrates how a keyboard handler may look like.

```
SafStatus MyKeyboard::onKeyboardInputChar( char c )
{
   switch ( c )
   {
    case 'D': debug(); break;
    case 'P': print(); break;
    case 'Q': quit();  break;
  }

   return Saf OK;
}
```

See Also: **SafKeyboard::onKeyboardInputLine**

SafKeyboard::onKeyboardInputLine

virtual SafStatus onKeyboardInputLine(const char* *input* **)**

Description

This virtual method is activated when the return key is pressed on the keyboard. The argument *input* is a single null terminated input line.

This function may be overloaded in sub-classes of **SafKeyboard**. Normally either **onKeyboardInputChar()** or **onKeyboardInputLine()** is overloaded, not both.

Access

This is a virtual protected method.

Parameters

input An input line from the keyboard.

Return Value

Should return *Saf_OK* on success, or an error code if the operation failed.

See Also: SafKeyboard::onKeyboardInputChar

SafKeyboard::SafKeyboard

`SafKeyboard()`

Description

This method is the constructor for the class **SafKeyboard**. It initialises the object.

Use the base class member method **activate()** to activate the keyboard handler.

Access

This constructor is public.

See Also: SafKeyboard::~SafKeyboard, SafHandler::activate

SafKeyboard::~SafKeyboard

`~SafKeyboard()`

Description

This method is the destructor for the class **SafKeyboard**. It destroys the *SWBus* object.

Access

This destructor is public.

See Also: SafKeyboard::SafKeyboard

SafLog

The class **SafLog** creates a file used by *SAF* for logging of error, debug or status messages.

Every time a *SAF* application is launched, a unique file is created. The file is created on a temporary directory generated using current time prefixed with the name SAFLOG. This file is available before entering the start-up entry point for the main program. The global **SafLog** instance *gSafLog* is created by *SAF* and is also available from global or static class constructors.

When entering the start-up point for the main program the generated file should be renamed using the member method **rename()**.

Note that the class **SafReport** uses the global instance *gSafLog* to report error, debug and status messages.

The class **SafLog** may be further sub-classed to add extra functionality. The attributes **mFile**, **mFileName** and the method **createLogFile()** are declared as protected in the class for this purpose.

Class Methods

Construction/Destruction	
SafLog	Constructs **SafLog** object.
~SafLog	Destroys **SafLog** object.

Operations	
createLogFile	Creates the log file.
put	Writes a message to the file.
rename	Renames the file.

Attributes	
mFile	Pointer to open file.
mFileName	The name of the file.

Member Functions

SafLog::createLogFile

```
SafStatus createLogFile()
```

Description

This method creates the log file. It uses the attribute **mFileName** when opening the file. The file is open for writing and appending. The stream pointer attribute **mFile** points to the open file. The attribute **mFile** is set to NULL if it fails to open the file.

This method is only of interest to developers that are sub-classing **SafLog**.

Access

This function is protected.

Return Value

Returns *Saf_OK* on success, otherwise the error code *Saf_Error*.

See Also: SafLog::SafLog

SafLog::put

SafStatus put(const char* *format*, … **)**

SafStatus put(SafStatus *status* **)**

Description

These methods write a message to the log file.

The method **put(const char*** *format*, **…)** writes a message to the log file accepting variable number of arguments. The parameter *format* uses the same formatting options used in the standard *C* function *printf*(). For a detailed description of these formatting options refer to a help file for *printf*().

The method **put(SafStatus** *status* **)** writes the message string associated with the **SafStatus** object to the file.

If writing messages to the display and to the log file, it is not recommended to use these methods. When using **put()** directly the message flow can not be changed at runtime. Instead use the **SafReport** macro **SafDMsg()**. For a detailed description refer to the **SafReport** macro sections.

Access

These functions are public.

Parameters

format	Format of the message.
…	Variable argument list.
status	**SafStatus** object.

Return Value

Returns *Saf_OK*.

See Also: SafReport, SafDMsg

SafLog::rename

SafStatus rename(const char* *filename* **)**

Description

This method renames and/or moves the log file.

Access

This function is public.

Parameters

filename The new name of the log file.

Return Value

Returns *Saf_OK* if the log file was renamed. *Saf_NullArg* is returned if the input argument is not a valid file name. *Saf_FileRenameFailed* is returned if the rename operation failed.

Example

```
main()
{
   gSafLog.rename( "app.log" );
   …
   return 0;
}
```

See Also: SafLog::put

SafLog::SafLog

SafLog()

SafLog(const char* *fileName* **)**

Description

The method **SafLog()** is the default class constructor. Note that this method does not initialise the attributes **mFileName** and **mFile**. This constructor is only of interest to developers that are sub-classing **SafLog**. In the sub-class constructor the attributes **mFileName** and **mFile** must be initialised.

The constructor **SafLog(const char*** *filename* **)** should be used when creating a **SafLog** instance, because it initialises the attributes and creates the log file.

Access

These constructors are public.

Parameters

filename The name of the log file.

Example

This example demonstrates how a default class constructor for a sub-class of **SafLog** may be implemented.

```
MyLog::MyLog() : SafLog()
{
   // Initialise the SafLog attributes.
   mFileName = NULL;
   mFile     = NULL;
}
```

See Also: SafLog::~SafLog

SafLog::~SafLog

virtual ~SafLog()

Description

This method is the destructor for the class **SafLog**. It flushes and closes the log file.

Access

This destructor is public.

See Also: **SafLog::SafLog**

Data Members

SafLog::mFile

Description

The attribute **mFile** points to the open file. If the file is not open this attribute is NULL.

Access

This attribute is protected.

SafLog::mFileName

Description

The attribute **mFileName** is the name of the file.

Access

This attribute is protected.

SafObjectStore

The class **SafObjectStore** is used as a container for managing lists of generic data objects. The data objects can be of any type.

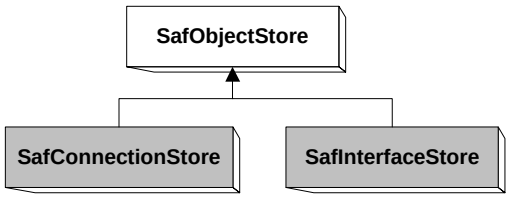
It provides functionality for adding, removing, traversing data objects in a double linked list.

This is an abstract class and must be further sub-classed to add functionality specific for objects of a class. It is not a template class hence only generic *void** are used as arguments to class members. Thus new methods should be created in sub-classes of **SafObjectStore** using the generic base class methods.

SafPOSITION is a data type used for accessing objects in the list. All objects added to the list are encapsulated in a **SafPOSITION** object. The data members in **SafPOSITION** should never be accessed outside member methods of **SafObjectStore**.

In *SAF* the classes **SafInterfaceStore** and **SafConnectionStore** are derived from **SafObjectStore**. These classes utilize functionality in the base class.

The example code used throughout the description of **SafObjectStore** is based on the class *VariableStore* that derives from **SafObjectStore**. The objects inserted into the **VariableStore** instance are of type *VariableObject*. The class definition for *VariableObject* is not of interest in the following example code and is therefore skipped.



Class Methods

Construction/Destruction	
SafObjectStore	Constructs **SafObjectStore** object.
~SafObjectStore	Destroys **SafObjectStore** object.

Pure Overridable Methods	
doDelete	Deletes the object passed as argument.
isEqual	Compares the two objects passed as arguments.

Operations	
doAppend	Appends a new object last in the list.
doAppendAt	Appends a new object at a specific position.
doInsert	Inserts a new object first in the list.
doInsertAt	Inserts a new object at a specific position.
doRemove	Removes an object from the list.
doRemoveAt	Removes an object from the list.

removeAll	Removes all objects from the list.

Get Methods

doGet	Returns a pointer to the object if it is in the list.
doGetHead	Returns a pointer to the first object in the list.
doGetNext	Returns a pointer to current object and sets the position to point to next object in the list.
doGetPrev	Returns a pointer to current object and sets the position to point to previous object.
doGetTail	Returns a pointer to the last object in the list.
getHeadPosition	Returns the position of the first object in the list.
getTailPosition	Returns the position of the last object in the list.
length	Returns number of objects in the list.

Data Types

SafPOSITION	Pointer to a position in the list.

Member Functions

SafObjectStore::doAppend

`SafTBool doAppend( void*` *`object`* **`)`**

Description

This method appends a new object last in the linked list.

Access

This is a protected method.

Parameters

object Pointer to the object to append to the list.

Return Value

Returns *SafTTrue* if the object was successfully appended to the list, or *SafTFalse* if the operation failed.

Example

```
SafTBool VariableStore::append( VariableObject* o )
{
   return doAppend( (void *)o );
}
```

See Also: **SafObjectStore::doAppendAt, SafObjectStore::doInsert, SafObjectStore::doInsertAt**

SafObjectStore::doAppendAt

SafTBool doAppend(void* *object*, **SafPOSITION&** *pos* **)**

Description

This method appends a new object at the position specified by the parameter *pos*.

Note that *pos* must be a reference to a valid position in the list.

Access

This is a protected method.

Parameters

object	Pointer to the object to append to the list.
pos	Position where the object should be appended.

Return Value

Returns *SafTTrue* if the object was successfully appended to the list, or *SafTFalse* if the operation failed.

Example

```
SafTBool VaiableStore::appendAt( VariableObject* o,
                                 SafPOSITION&    pos )
{
   return doAppendAt( (void *)o, pos );
}
```

See Also: **SafObjectStore::doAppend, SafObjectStore::doInsert, SafObjectStore::doInsertAt**

SafObjectStore::doDelete

virtual void doDelete(void* *object* **) =0**

Description

This pure virtual method must be overloaded in sub-classes of **SafObjectStore**. It should delete the object passed as parameter.

If the object was allocated using the operator *new* the object should be deleted using the operator *delete*. Remember to cast the object to the correct class before deleting the object.

Note that this method is automatically called within **SafObjectStore** when objects are deleted. It should never be necessary to call this method from any sub-class of **SafObjectStore**.

Access

This is a pure virtual protected method. Must be implemented in sub-classes of **SafObjectStore**.

Parameters

object	The object to delete.

Example

Note that the object is cast to the class *VariableObject*.

```
void VariableStore::doDelete( void* object )
{
   delete (VariableObject *)object;
}
```

See Also: **SafObjectStore::removeAll, SafObjectStore::doRemove, SafObjectStore::doRemoveAt**

SafObjectStore::doGet

`void* doGet( void*` *`object`* `) const`

Description

This method checks if the object exists in the list. If the object exists a pointer to the object is returned, otherwise NULL.

Access

This is a protected method.

Parameters

object Pointer to the object to check if exist in the list.

Return Value

Returns a pointer to the object if it exists in the list, otherwise NULL.

Example

```
VariableObject* VariableStore::get( VariableObject* o ) const
{
   return (VariableObject *)doGet( (void *)o );
}
```

See Also: **SafObjectStore::doGetHead, SafObjectStore::doGetTail**

SafObjectStore::doGetHead

`void* doGetHead() const`

Description

This method returns a pointer to the first object in the list.

Access

This is a protected method.

Return Value

Returns a pointer to the first object in the list. If the list is empty NULL is returned.

Example

```
VariableObject* VariableStore::getHead( VariableObject* o ) const
{
   return (VariableObject *)doGetHead( (void *)o );
}
```

See Also: **SafObjectStore::doGetTail**

SafObjectStore::doGetNext

void* doGetNext(SafPOSITION& *pos* **) const**

Description

This method returns a pointer to the object at current position given by the parameter *pos*.

The parameter *pos* is set to point to the next object in the list.

Note that the parameter *pos* must represent a valid position in the list. It must have been initialised using the method **getHeadPosition()** or **getTailPosition()**.

Access

This is a protected method.

Parameters

pos	A reference to a valid position in the list. The position is set to point to the next object in the list. When reaching the end of the list the value of this variable is NULL. Use NULL as a termination condition in a loop.

Return Value

Returns a pointer to the current object in the list. NULL is returned if the input parameter is NULL.

Example

This first example demonstrates how to implement a typical **getNext()** method. The second example demonstrates the use of **getNext()** to traverse the list of objects.

```
VariableObject* VariableStore::getNext( SafPOSITION& pos ) const
{
   return (VariableObject *)doGetNext( pos );
}
```

```
void GetAllVariables( VariableStore& store )
{
   SafPOSITION pos = store.getHeadPosition();

   while ( pos )   /* As long as pos is true */
   {
      VariableObject* vO = store.getNext( pos );
      …            /* Do something with vO */
   }
}
```

See Also: **SafObjectStore::doGetPrev, SafObjectStore::getHeadPosition, SafObjectStore::getTailPosition**

SafObjectStore::doGetPrev

void doGetPrev(SafPOSITION& *pos* **) const**

Description

This method returns a pointer to the object at current position given by the parameter *pos*.

The parameter *pos* is set to point to the previous object in the list.

Note that the parameter *pos* must represent a valid position in the list. It must have been initialised using the method **getHeadPosition()** or **getTailPosition()**.

Access

This is a protected method.

Parameters

pos	A reference to a valid position in the list. The position is set to point to the previous object in the list. When reaching the first element in the list the value of this variable is NULL. Use NULL as a termination condition in a loop.

Return Value

Returns a pointer to current object in the list. NULL is returned if the input parameter is NULL.

Example

See example for **doGetNext()**.

See Also: **SafObjectStore::doGetPrev, SafObjectStore::getTailPosition, SafObjectStore::getHeadPosition**

SafObjectStore::doGetTail

```
void* doGetTail() const
```

Description

This method returns a pointer to the last object in the list

Access

This is a protected method.

Return Value

Returns a pointer to the last object in the list. If the list is empty, NULL is returned.

Example

See example for **doGetHead()**.

See Also: **SafObjectStore::doGetHead**

SafObjectStore::doInsert

```
SafTBool doInsert( void* object )
```

Description

This method inserts the object passed as argument first in the list.

Access

This is a protected method.

Parameters

object	Object to insert into the list.

Return Value

Returns *SafTTrue* if the object was successfully inserted into the list, or *SafTFalse* if the operation failed.

Example

See example for **doAppend()**.

See Also: **SafObjectStore::doInsertAt, SafObjectStore::doAppend, SafObjectStore::doAppendAt**

SafObjectStore::doInsertAt

```
SafTBool doInsertAt( void* object, SafPOSITION& pos )
```

Description

This method inserts the object passed as argument at the position given by the parameter *pos.*

Note that *pos* must be a reference to a valid position in the list.

Access

This is a protected method.

Parameters

object	Pointer to object to insert into the list.
pos	Where to insert the object. Must be a valid position in the list.

Return Value

Returns *SafTTrue* if the object was successfully inserted into the list, or *SafTFalse* if the operation failed.

Example

See example for **doAppendAt()**.

See Also: **SafObjectStore::doInsert**

SafObjectStore::doRemove

```
SafTBool doRemove( void*    object,
                   SafTBool doDeleteElement = SafTTrue )
```

Description

This method removes the object specified as argument from the list. If the parameter *doDeleteElement* is true the method **doDelete()** is invoked to delete the object.

Access

This is a protected method.

Parameters

object	The object to delete.
doDeleteElement	*SafTTrue* if the object should be deleted, *SafTFalse* if not.

Return Value

Returns *SafTTrue* if the object was successfully removed from the list, or *SafTFalse* if the operation failed.

Example

```
SafTBool VariableStore::remove( VariableObject* o )
{
   return doRemove( (void *)o, SafTTrue );
}
```

See Also: **SafObjectStore::doDelete, SafObjectStore::doRemoveAt**

SafObjectStore::doRemoveAt

```
SafTBool doRemoveAt( SafPOSITION& pos,
                     SafTBool     doDeleteElement = SafTTrue )
```

Description

This method removes the object at position *pos* from the list. The parameter *pos* must represent a legal list position. If the parameter *doDeleteElement* is true the method **doDelete()** is invoked to delete the object.

Access

This is a protected method.

Parameters

pos	Position to the object to remove from the list.
doDeleteElement	*SafTTrue* if the object should be deleted, *SafTFalse* if not

Return Value

Returns *SafTTrue* if the object was successfully removed from the list, or *SafTFalse* if the operation failed.

Example

```
SafTBool VariableStore::removeAt( SafPOSITION& pos )
{
   return doRemoveAt( pos, SafTFalse );
}
```

See Also: **SafObjectStore::doDelete, SafObjectStore::doRemove**

SafObjectStore::getHeadPosition

```
SafPOSITION getHeadPosition() const
```

Description

This method returns the position to the first element in the list.

Access

This function is public.

Return Value

Returns the position to the first element in the list. NULL if the list is empty.

Example

See example for **doGetNext()**.

See Also: **SafObjectStore::getTailPosition, SafObjectStore::doGetNext, SafObjectStore::doGetPrev**

SafObjectStore::getTailPosition

```
SafPOSITION getTailPosition() const
```

Description

This method returns the position to the last element in the list.

Access

This function is public.

Return Value

Returns the position to the last element in the list. NULL if the list is empty.

Example

See example for **doGetNext()**.

See Also: **SafObjectStore::getHeadPosition, SafObjectStore::doGetNext, SafObjectStore::doGetPrev**

SafObjectStore::isEqual

```
virtual SafTBool isEqual( const void* o1, const void* o2 ) const =0
```

Description

This pure virtual method must be overloaded in sub-classes of **SafObjectStore**. It should compare whether the objects *o1* and *o2* are equal or not.

Note that the objects must be cast to the correct data type before they are being compared.

Access

This is a pure virtual protected method. Must be implemented in sub-classes of **SafObjectStore**.

Parameters

o1	Pointer to first object.
o2	Pointer to second object.

Return Value

Must returns *SafTTrue* if the objects are equal, otherwise *SafTFalse*.

Example

These examples demonstrates two different implementations of the method **isEqual()**. The first implementation only checks if the objects are the same. The second example uses the *VariableObject* method *name()* to check if the objects are equal.

```
SafTBool VariableStore::isEqual( const void* o1,
                                 const void* o2 ) const
{
   return o1 == o2;
}
```

```
SafTBool VariableStore::isEqual( const void* o1,
                                 const void* o2 ) const
{
   return !strcmp( ((VariableObject *)o1)->name(),
                   ((VariableObject *)o2)->name() );
}
```

See Also: **SafObjectStore::doDelete**

SafObjecStore::length

`int32 length() const`

Description

This method returns the number of objects put into the list.

Access

This function is public.

Return Value

Returns number of objects in the list.

See Also: **SafObjectStore::doAppend, SafObjectStore::doAppendAt, SafObjectStore::doInsert, SafObjectStore::doInsertAt**

SafObjectStore::removeAll

`void removeAll( SafTBool` *`deleteElements`* **`= SafTTrue )`**

Description

This method removes all the objects from the list. If the parameter *deleteElements* is true the method **doDelete()** is invoked for each object in the list.

Note that the destructor for **SafObjectStore** does not remove all the objects in the list. A good practice is thus to call this method in the destructor of the sub-class of **SafObjectStore**

Access

This function is public.

Parameters

deleteElements *SafTTrue* if the objects in the list should be deleted, otherwise *SafTFalse*.

Example

```
VariableStore::~VariableStore()
{
    removeAll( SafTTrue );
}
```

See Also: **SafObjectStore::doRemove, SafObjectStore::doRemoveAt, SafObjectStore::doDelete**

SafObjectStore::SafObjectStore

```
SafObjectStore()
```

Description

This is the constructor for the class **SafObjectStore**. It initialises the member attributes.

Access

This constructor is public.

Example

See Also: **SafObjectStore::~SafObjectStore**

SafObjectStore::~SafObjectStore

```
virtual ~SafObjectStore()
```

Description

This is the destructor for the class **SafObjectStore**.

Note that this destructor does not remove the objects in the list. The destructors of the sub-classes must remove all the objects by calling the method **removeAll()**.

Access

This destructor is public.

Example

See the example for **removeAll()**.

See Also: **SafObjectStore::SafObjectStore**

Data Types

SafPOSITION

Description

SafPOSITION is a data type used when accessing objects in any class derived from the base class **SafObjectStore**. It can for instance be used to traverse the objects in the list. A variable of this type must always be pointing to a legal list position before it is used. Use either the method **getHeadPosition()** or **getTailPosition()** to initialize the variable.

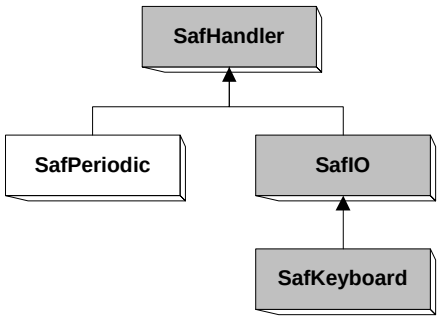
All objects added to the list are encapsulated in a **SafPOSITION** object. Notice that the data members in **SafPOSITION** are inaccessible outside class member methods of **SafObjectStore**.

SafPeriodic

The class **SafPeriodic** is used when creating periodic objects in *SAF*. A periodic object is called regularly from the main loop in *SAF*.

The class **SafPeriodic** must be further sub-classed and the virtual methods **onAction()** and **onTerminate()** may be overloaded.

To activate a periodic object the base class member method **activate()** must be called.



Class Methods

Construction/Destruction	
SafPeriodic	Constructs **SafPeriodic** object.
~SafPeriodic	Destroys **SafPeriodic** object.

Operations	
setInterval	Changes the periodic interval.

Overridable Methods	
onTerminate	Called when the periodic object terminates.

Get Methods	
timeSinceLastInvocation	Time between invocations of the **onAction()** method.
timeSinceThisInvocation	Time used so far in the **onAction()** method.
timeSpentInLastInvocation	Time used in the **onAction()** method last time it was called.

Member Functions

SafPeriodic::onTerminate

```
virtual SafStatus onTerminate()
```

Description

This virtual method is automatically called when the time specified for the periodic object expires.

This period must be specified in the parameter *period* in the class constructor. It will not be called if the parameter *period* is set to –1. For a detailed description refer to the class constructor **SafPeriodic**.

Access

This is a virtual protected method.

Return Value

Must return *Saf_OK* on success, or an error code if it failed.

See Also: SafPeriodic::SafPeriodic

SafPeriodic

```
SafPeriodic( int32 interval, int32 count = -1, int32 period = -1 )
```

Description

This is the constructor for the class **SafPeriodic**. It initialises the periodic object.

Note that the **SafPeriodic** sub-class instance is not activated before calling the base class method **activate()**. The periodic object can at any time be suspended or restarted.

The parameters *count* and *period* are optional (default values).

The conditions can at any time be changed for the periodic object by calling the method **setInterval()**.

Access

This constructor is public.

Parameters

interval	Specifies the interval in milliseconds between calls to the method **onAction()**.
count	Number of times the method **onAction()** will be called before the periodic object is terminated. If set to –1, there is no such limit.
period	Specifies the interval in milliseconds to wait before calling the method **onTerminate()**. If set to –1, the method **onTerminate()** will never be called.

Example

This example demonstrates how a **SafPeriodic** sub-class (*MyPeriodic*) may be created. The **onAction()** method will be called every fifth second for a period of one hour. When the hour has expired the method **onTerminate()** will be called.

```
MyPeriodic::MyPeriodic() : SafPeriodic( 5000, -1, 360000 )
{
   activate();  // Start the periodic object
}

SafStatus MyPeriodic::onAction()
{
   /* Periodic invocation */
   return Saf_OK;
}

SafStatus MyPeriodic::onTerminate()
{
   /* The periodic object has terminated. */
   return Saf OK;
}
```

See Also: **SafPeriodic::~SafPeriodic, SafHandler::onAction, SafPeriodic::onTerminate**

SafPeriodic::setInterval

`SafStatus setInterval( int32 `*`interval`*` )`

Description

This method changes the periodic interval for the invocation of the method **onAction()**. Note that the object must be activated before calling this method.

The periodic object is set to a suspended state after calling this method.

Access

This function is public.

Parameters

interval	Specifies the time in milliseconds between each call to the method **onAction()**.

Return Value

Returns *Saf_OK* if the operation completed successfully. *Saf_SbNotActive* is returned if the object is not activated. If the operation failed *Saf_InternalError* is returned.

See Also: **SafPeriodic::SafPeriodic, SafHandler::activate**

SafPeriodic::timeSinceLastInvocation

`uint32 timeSinceLastInvocation() const`

Description

This method returns the time in milliseconds between last and current invocation of the method **onAction()**.

Access

This is a protected method.

Return Value

Returns the time in milliseconds between invocations of the method **onAction()**.

See Also: **SafPeriodic::timeSinceThisInvocation,**
SafPeriodic::timeSpentInLastInvocation

SafPeriodic::timeSinceThisInvocation

```
uint32 timeSinceThisInvocation() const
```

Description

This method returns the time used in the method **onAction()** since it was invoked. The time returned is in milliseconds.

Access

This is a protected method.

Return Value

Returns the time (in milliseconds) spent so far in the method **onAction()**.

See Also: **SafPeriodic::timeSinceLastInvocation,**
SafPeriodic::timeSpentInLastInvocation

SafPeriodic::timeSpentInLastInvocation

```
uint32 timeSpentInLastInvocation() const
```

Description

This method returns the time spent in the last invocation of the method **onAction()**. The time returned is in milliseconds.

Access

This function is public.

Return Value

Returns the time (in milliseconds) spent in the last invocation of the method **onAction()**.

Example

This example uses the time methods to measure the time spent in, used so far and between invocations of the method **onAction()**.

```
SafStatus MyPeriodic::onAction()
{
   cout << "Time since last onAction method was called = "
        << timeSinceLastInvocation() << "ms." << endl;
   cout << "Time spent in last onAction method = "
        << timeSpentInLastInvocation() << " ms." << endl;

   doSomething 1();
   cout << "Time used in onAction after doSomething 1() = "
        << timeSinceThisInvocation() << " ms." << endl;
   doSomething 2();
   cout << "Time used in onAction after doSomething 2() = "
        << timeSinceThisInvocation() << " ms." << endl;

   return Saf OK;
}
```

See Also: **SafPeriodic::timeSinceLastInvocation, SafPeriodic::timeSinceThisInvocation**

SafPeriodic::~SafPeriodic

`~SafPeriodic()`

Description

This is the destructor for the class **SafPeriodic**. It destroys the *SWBus* object.

Access

This destructor is public.

See Also: **SafPeriodic::SafPeriodic**

SafReadScript

SafReadScript is a class used for reading database (pdat) files.

This class is very well suited for creating *SWBus* classes and variables. It is not necessary to use this class if the only goal is to create *SWBus* classes and variables without doing anything with the objects. Consider using **SbReadScript()** instead. But to achieve increased flexibility and functionality not found in **SbReadScript()** the class should be further sub-classed and certain virtual methods should be overloaded.

The following methods are virtual in **SafReadScript**:

- **classCreated**
- **existingVarRead**
- **createVar**

Memory is allocated for each of the variables read from the database file. The data pointer argument used in the virtual method **createVar()** must be deleted using the standard function **free()** if a variable uses another data area.

If there are any syntax errors in the database file, the line numbers where the errors occurred will be printed. The method **numErrors()** will return the number of errors encountered in the database file.

Class Methods

Construction/Destruction	
SafReadScript	Constructs **SafReadScript** object.
~SafReadScript	Destroys **SafReadScript** object.

Overridable Methods	
classCreated	Called after an *SWBus* class has been created.
createVar	Called to create an *SWBus* variable.
existingVarRead	Called when trying to create an *SWBus* variable that exists.

Operators	
parse	Starts parsing the specified database file.

Get Methods	
numErrors	Returns number of errors in the database file.

Member Functions

SafReadScript::classCreated

```
virtual void classCreated( SbTSTI theClass )
```

Description

This virtual method is called after an *SWBus* class has been created.

To achieve this functionality the method must be overloaded in sub-classes of **SafReadScript**.

Access

This is a virtual protected method.

Parameters

theClass	The *SWBus STI* of the created class.

See Also: **SafReadScript::parse**

SafReadScript::createVar

```
virtual SbTSTI createVar( SbTSTI      theClass,
                          const char* name,
                          void*       data )
```

Description

This virtual method should create the *SWBus* variable. Overload this method in the sub-class of **SafReadScript**.

The parameter *data* points to a data area allocated in the *SWBus* using the standard function *malloc()*. If this data area is not needed when creating the *SWBus* variable delete the area using the standard function *free()*.

Access

This is a virtual protected method.

Parameters

theClass	The class of the new variable to create.
name	Name of the new variable to create.
data	Pointer to a data area created in the *SWBus*.

Return Value

Should return the *SWBus STI* of the new object, or *SbCError* if the operation failed.

Example

The first example shows the default implementation of **createVar()** in **SafReadScript**.

```
SbTSTI SafReadScript::createVar( SbTSTI theClass,
                                 const char* name,
                                 void*       data )
{
   return SbCreate( SbCPLocal, theClass, name, SbCNull, data );
}
```

The next example demonstrates the use of **createVar()** to change the data pointers of *SWBus* variables using a user defined data area. The implementation of the function **getDataPtr()** is not essential in this example. It just returns a pointer to the data area for the variable passed as argument. Note that error checking has been skipped for simplicity.

```
SbTSTI ReadScript::createVar( SbTSTI      theClass,
                              const char* name,
                              void*       data )
{
   free( data );               // Free data allocated in the SWBus

   data = getDataPtr( name );  // Get user defined data area.

   return SbCreate( SbCPLocal, theClass, name, SbCNull, data );
}
```

See Also: **SbCreate**

SafReadScript::existingVarRead

virtual void existingVarRead(SbTSTI *theVar* **)**

Description

This virtual method is called when a variable that already exists in the namespace of the *SWBus* is trying to be created.

To get this functionality overload this method in the sub-class of **SafReadScript**.

Access

This is a virtual protected method.

Parameters

theVar *SWBus STI* of the existing variable with the same name.

See Also: **SafReadScript::parse**

SafReadScript::numErrors

uint32 numErrors() const

Description

This method returns the number of errors produced when parsing the last database file.

Access

This function is public.

Return Value

Returns the number of errors.

See Also: **SafReadScript::parse**

SafReadScript::parse

```
SafStatus parse( const char* filename )
```

Description

This method starts the parsing of the database file passed as argument.

Note that only one parsing can be executed simultaneously.

Access

This function is public.

Parameters

filename Name of the database file to parse.

Return Value

Returns *Saf_OK* if the file was successfully parsed. If errors where encountered during the parsing *Saf_RSParserError* is returned. If the filename passed as argument is not a valid filename *Saf_IllegalFileName* is returned. If the method **parse()** has been invoked *Saf_RSInvoked* is returned.

See Also: **SafReadScript::numErrors, SbReadScript**

SafReadScript::SafReadScript

```
SafReadScript()
```

Description

This is the constructor for the class **SafReadScript**. It initialises the data members.

Access

This constructor is public.

See Also: **SafReadScript::~SafReadScript**

SafReadScript::~SafReadScript

```
virtual ~SafReadScript()
```

Description

This is the destructor for the class **SafReadScript**.

Access

This destructor is public.

See Also: **SafReadScript::SafReadScript**

SafReport

SafReport is a class used for printing messages to an output device and a log file. Different messages can be turned on and off in the application using **SafReport** functionality. **SafReport** can be used for pinpointing problems in an application. It can simplify and reduce the time and expenses attended with debugging an application.

Messages can be turned on or off in an application either offline or online. The amount of output messages a process produce can be changed offline either by command line options or by changes in the source code. Another very powerful feature is that the external interface class **SafReportExternalInterface** can be used to change the message flow online in any *SAF* compliant process.

The class **SafReport** is available for developers but is not documented because it is not recommended to access **SafReport** methods directly in the application source code. *SAF* offers instead a set of macros that should be used.

There are two different types of **SafReport** objects, categories and contexts. A category is used to separate different message types, while a context is used to indicate the state the program is currently running in. Macros are available to make sure that all **SafReport** objects are initialised before entering the main entry point in the program, i.e. that categories and contexts can be used in global class constructors.

When adding **SafReport** objects these must be declared in a header (*.h) file, defined and properties must be added in a **SafReport** definition block in a source (*.cpp, *.C, *.cxx) file. Macros are available to lighten and to ensure that they are added in accordance with the guidelines for adding such **SafReport** objects. A complete example is demonstrated in **SafDDeclareReport**. Refer to this example for details on how to add **SafReport** objects.

SafReport divides messages in different categories. A category is a message type. For instance debug, error and flow are examples of different categories. The following predefined categories are available in *SAF*:

- **SAF_CAT_ERROR**
- **SAF_CAT_WARNING**
- **SAF_CAT_DEBUG**
- **SAF_CAT_INFO**
- **SAF_CAT_FLOW**

New categories can be added to an application by using the macros **SafDDeclareCategory()**, **SafDImplementCategory()** and **SafDAddCategory()**. The numbers of categories that can be defined in *SAF* are limited. In current version of *SAF* this limit is 32. In practice is this not a restriction but a way to limit the number of categories. If defining too many categories it may be difficult to use correct categories in the source code.

When defining a category, a message format must be specified. Information like time, file name, line number, class name and function name can be specified using directives. A directive in the format string is the character ‘%’ followed by another character specifying the type.

All categories belong to a group. *SAF* has a set of predefined groups. A group identifier is used to group messages in different categories. **SafDClassDebug** is an example of a predefined group identifier. All debug messages should use **SafDClassDebug** as group identifier. This information can be used to prevent that error messages are turned off for instance when using the external interface **SafReportExternalInterface** from a remote process.

SafReport also has a concept called a context. When printing a message in an application it is always in a certain context. The application can for instance be in an initialisation phase or in a subscription

phase. Those are two examples of different contexts. When using the powerful features of contexts, another process can turn on and off messages in certain parts (contexts) of the code for debugging purposes.

The following contexts are predefined in *SAF*:

- **SAF_CON_SWBUS**
- **SAF_CON**
- **SAF_CON_FLOW**

New contexts should be created when developing an application. The macros **SafDDeclareContext()**, **SafDImplementContext()** and **SafDAddContext()** are used for this purpose.

All classes should use the macros **SafDDeclareClass** and **SafDImplementClass()**. The information stored in these macros are used to get the class name when printing messages from member functions.

All functions in an application should use the macro **SafDFunction()**. The first statement in every function should be this macro. When printing a message within a method this information is used by the macro **SafDMsg()** to get the function name. If the pre-processor symbol **SAF_FLOW_LOG** is defined when compiling the application source code, flow information is added as well. Be careful using this option when compiling because it reduces performance.

The macro **SafDMsg()** is used to print a message. Two of the arguments to this macro are the context and the category. Never use *printf()*, *cout*, *cerr*, *fprintf()*, *puts()* or any other function printing messages to an output device. Always use **SafDMsg()**.

SafReport can be regarded as a two dimensional matrix where the categories are columns and the contexts are rows. Entries in this matrix can be enabled and disabled. Only entries that are enabled will be printed by the macro **SafDMsg()**.Two macros are available for enabling and disabling entries in this matrix, these are **SafDEnableOutput()** and **SafDDisableOutput()**.

Macros

Information	
SafDDeclareClass	Adds information to a class used by **SafReport**.
SafDFunction	Adds information to a method used by **SafReport**.
SafDImplementClass	Implements information for a class used by **SafReport**.

Declaration	
SafDDeclareCategory	Declares a new **SafReport** category.
SafDDeclareContext	Declares a new **SafReport** context.
SafDDeclareReport	Ensures that **SafReport** objects are initialised.

Definitions	
SafDImplementCategory	Defines a new **SafReport** category.
SafDImplementContext	Defines a new **SafReport** context.
SafDImplementReportBegin	Marks the beginning of **SafReport** definition block.
SafDImplementReportEnd	Marks the end of **SafReport** definition block.

Properties	
SafDAddCategory	Adds properties to a **SafReport** category object.

SafDAddContext	Adds properties to a **SafReport** context object.

Operations

SafDChangeCategoryFormat	Changes the format of a **SafReport** category object.
SafDDisableOutput	Disables **SafReport** category/context pair.
SafDEnableOutput	Enables **SafReport** category/context pair.

Output

SafDMsg	Prints a message to an output device and log file.

Macros

SafDAddCategory

SafDAddCategory(*CATEGORY*, *NAME*, *FORMAT*, *GROUPID*, *ENABLE* **)**

Description

This macro is used to add properties to a category.

The argument *CATEGORY* is the identification of the category. The same identification must be passed as argument to the macros **SafDDeclareCategory()** and **SafDImplementCategory()**.

The argument *NAME* is a human readable name. Should be more descriptive than the argument *CATEGORY*.

Associated with a category is a format string. It is specified in the *FORMAT* argument. This format string defines what kind of information to print. It is an ordinary string with directives. A directive is a '%' character followed by one of the characters:

- **'%'** Prints the '%' character.
- **'Y'** Prints the year.
- **'M'** Prints the month.
- **'D'** Prints the day.
- **'H'** Prints the hour.
- **'m'** Prints the minute.
- **'S'** Prints the second.
- **'s'** Prints the time in milliseconds.
- **'L'** Prints the line number where the message was put.
- **'F'** Prints the file where the message was put.
- **'T'** Prints the category.
- **'N'** Prints the context.
- **'C'** Prints the class if the message was put within a member method.
- **'f'** Prints the name of the function where the message was put.
- **'t'** Prints the message.

The argument *GROUPID* is used to group categories belonging together. For instance it can be very useful to add different debug levels in an application. These categories will have group identifier **SafDClassDebug**. The following group identifiers are predefined in *SAF*:

- **SafDClassMessage**
- **SafDClassInfo**
- **SafDClassError**
- **SafDClassWarning**
- **SafDClassDebug**
- **SafDClassFlow**
- **SafDClassMemory**
- **SafDClassComm**
- **SafDClassData**

It is not necessary to use any of these predefined groups when adding a category. User defined group identifiers can be used. Note that group identifiers in the range 0..100 (inclusive) are reserved for future use by *SAF*.

Note that **SafDAddCatetgory()** must be put inside a **SafDImplementReportBegin()** and **SafDImplementReportEnd()** block.

Parameters

CATEGORY	The category identification.
NAME	A human readable name of the category.
FORMAT	Format of the message to print.
GROUPID	What group this category belongs to.
ENABLE	Whether this category is default enabled or disabled. It must have the value *SafCEnable* or *SafCDisable*.

Example

This example will demonstrate how to define a category, how it is used and the result of using it. Note that this is not a complete example. For a complete example refer to the description of the macro **SafDDeclareReport()**.

```
…
SafDAddCategory( MY CAT,
                 "My Category",
                 "%Y.%M.%D %H:%m:%S %F.%L %C::%f\n%T %N: %t",
                 SafDClassMessage,
                 SafCEnable );
…
```

The following code is an example using the category defined above. This implementation just prints the address of the *this* pointer.

```
void MyClass::MyFunc()
{
   SafDFunction( MyFunc );

   SafDMsg( MY CAT, APP CON, ( "This pointer = %p\n", this ) );
}
```

The output produced by the macro **SafDMsg()** may look something like.

```
…
2000.10.24 14:32:23 MyFile.h.1234 MyClass::MyFunc
MY CAT APP CON: This pointer = 0x31454680
…
```

See Also: **SafDDefineCategory, SafDImplementCategory, SafDFunction, SafDMsg**

SafDAddContext

SafDAddContext(*CONTEXT*, *NAME*, *DESCRIPTION* **)**

Description

This macro is used to add properties to a context.

The argument *CONTEXT* is the identification of the context. The same identification must be passed as argument to the macros **SafDDeclareContext()** and **SafDImplementContext()**.

The argument *DESCRIPTION* should describe what the purpose is with the context.

Note that this macro must be put inside a **SafDImplementReportBegin()** and **SafDImplementReportEnd()** block.

Parameters

CONTEXT The context identification.

NAME A human readable name of the context.

DESCRIPTION A long readable description of the context.

Example

For a complete example refer to the description of the macro **SafDDeclareReport()**.

See Also: **SafDDeclareContext, SafDImplementContext**

SafDChangeCategoryFormat

SafDChangeCategoryFormat(*CATEGORY*, *FORMAT*, *ENABLE* **)**

Description

This macro is used to change the format string of a category.

This macro is normally used to change the format string of one of the predefined categories.

For more information about the format string refer to **SafDAddCategory()**.

Note that this macro must be put inside a **SafDImplementReportBegin()** and **SafDImplementReportEnd()** block.

Parameters

CATEGORY	The category identification.
FORMAT	Format of the message to print.
ENABLE	Whether this category is default enabled or disabled. It must have the value *SafCEnable* or *SafCDisable*

Example

For a complete example refer to the description of the macro **SafDDeclareReport()**.

See Also: SafDAddCategory

SafDDeclareCategory

SafDDeclareCategory(*CATEGORY* **)**

Description

This macro declares a category. This macro must be put in a header file.

The argument *CATEGORY* is the name of the category. The same name must be passed as argument to the macros **SafDImplementCategory()** and **SafDAddCategory()**.

Parameters

CATEGORY	The category identification.

Example

For a complete example refer to the description of the macro **SafDDeclareReport()**.

See Also: SafDImplementCategory, SafDAddCategory

SafDDeclareClass

SafDDeclareClass

Description

This macro is used inside a class definition to add information used by the macro **SafDMsg()** when a class method outputs a message. The macro **SafDImplementClass()** must be used outside the class definition to define the variable declared in this macro. The information stored in this macro is the name of the class.

The category format directive '%C' is used to print the class name. For more information refer to **SafDAddCategory()**.

This macro should be put inside every class.

Example

This is an example of a class using the macros **SafDDeclareClass** and **SafDImplementClass()**.

```
#ifndef   MY CLASS H
#define   MY CLASS H

// Filename: MyClass.h

#include <Saf.h>

class MyClass
{
  …
  private:
     MyClass();

   …

   SafDDeclareClass;
};

#endif // __MY_CLASS_H__
```

The implementation of the class *MyClass* in the *C* file.

```
#include "MyClass.h"


SafDImelementClass( MyClass );


MyClass::MyClass()
{
   SafDFunction( MyClass );

   …

   SafDMsg( SAF CAT DEBUG, APP CON, ( "In constructor\n" ) );
}
```

See Also: **SafDImplementClass, SafDFunction, SafDMsg**

SafDDeclareContext

SafDDeclareContext(*CONTEXT* **)**

Description

This macro declares a context. This macro must be put in a header file.

The argument *CONTEXT* is the identification of the context. The same identification must be passed as argument to the macros **SafDImplementContext()** and **SafDAddContext()**.

Parameters

CONTEXT The context identification.

Example

For a complete example refer to the description of the macro **SafDDeclareReport()**.

See Also: **SafDImplementContext, SafDAddContext**

SafDDeclareReport

SafDDeclareReport(*NAME* **)**

Description

This macro must be used in all header files defining **SafReport** objects using the macros **SafDDeclareCategory()** or **SafDDeclareContext()**. It adds functionality for ensuring that all objects are initialised before any constructors for global variables are called and the main entry point in the application is entered.

Note that the name passed as argument to this macro must be unique. The same name must be passed as argument to the macros **SafDImplementReportBegin()** and **SafDImplementReportEnd()**.

Parameters

NAME	Unique name used to identify **SafReport** definition section in the *C* file.

Example

This is a complete example demonstrating how to add new categories and contexts in an application. The following categories are added, *DEBUG_LEVEL_1* and *DEBUG_LEVEL_2*. The following contexts are added, *APP_INIT*, *APP_SUBS* and *APP_SEND*.

The header file declares the categories and the contexts.

```
#ifndef   APP REP INIT H
#define __APP_REP_INIT_H__

#include <Saf.h>

// Filename: AppRepInit.h

// Declare categories
SafDDeclareCategory( DEBUG LEVEL 1 );
SafDDeclareCategory( DEBUG LEVEL 2 );

// Declare contexts
SafDDeclareContext( APP INIT );
SafDDeclareContext( APP SUBS );
SafDDeclareContext( APP SEND );


SafDDeclareReport(   APP REP INIT   );

#endif // __APP_REP_INIT_H__
```

The following code declares and adds properties to the categories and contexts declared in the header file *"AppRepInit.h"*. Note the use of **SafDChangeCategoryFormat()** to change the format of a predefined *SAF* category.

```
#include "AppRepInit.h"


// Define the categories
SafDImplementCategory( DEBUG LEVEL 1 );
SafDImplementCategory( DEBUG_LEVEL_2 );

// Define the contexts
SafDImplementContext( APP_INIT );
SafDImplementContext( APP SUBS );
SafDImplementContext( APP SEND );


// Define format string for debug categories
static char* sDebugFormat = "%H:%m:%S %F.%L %C%f\n%T %N: %t"


// Add properties to the categories and contexts
SafDImplementReportBegin(   APP REP INIT   );

   // Categories
   SafDAddCategory( DEBUG LEVEL 1, "Debug Level 1", sDebugFormat,
                    SafDClassDebug, SafCEnable );
   SafDAddCategory( DEBUG LEVEL 2, "Debug Level 2", sDebugFormat,
                    SafDClassDebug, SafCEnable );

   // Contexts
   SafDAddContext( APP INIT, "AppInititialisation",
                   "Initialisation of the application" );
   SafDAddContext( APP SUBS, "AppSubscription",
                   "Subscription to new variables" );
   SafDAddContext( APP SEND, "AppSend",
                   "Sending variables to external process" );

   // Change format for the predefined category SAF_CAT_DEBUG
```

```
   SafDChangeCategoryFormat( SAF_CAT_DEBUG, sDebugFormat,
                             SafCEnable );

SafDImplementReportEnd( __APP_REP_INIT__ );
```

See Also: **SafDImplementReportBegin, SafDImplementReportEnd, SafDAddContext, SafDAddCategory, SafDDeclareContext, SafDDeclareCategory, SafDImplementContext, SafDImplementCategory, SafDChangeCategoryFormat**

SafDDisableOuptut

SafDDisableOutput(*CATEGORY*, *CONTEXT* **)**

Description

This macro disables the corresponding category and context entry in the **SafReport** matrix.

If the argument *CONTEXT* is –1 all the contexts for the specified category will be disabled.

If the argument *CATEGORY* is –1 all the categories for the specified context will be disabled.

If an entry in the **SafReport** matrix is disabled no output will be printed for this category and context.

Parameters

CATEGORY The category identification.

CONTEXT The context identification.

Example

This example disables all messages for the context *APP_CON_INIT*.

```
main()
{
   SafDDisableOutput( -1, APP CON INIT );
   …
}
```

See Also: **SafDEnableOutput**

SafDEnableOutput

SafDEnableOutput(*CATEGORY*, *CONTEXT* **)**

Description

This macro enables the corresponding category and context entry in the **SafReport** matrix.

If the argument *CONTEXT* is –1 all the contexts for the specified category will be enabled.

If the argument *CATEGORY* is –1 all the categories for the specified context will be enabled.

If an entry in the **SafReport** matrix is enabled output will be printed for this category and context.

Parameters

CATEGORY The category identification.

CONTEXT The context identification.

See Also: **SafDDisableOutput**

SafDFunction

SafDFunction(*FUNCTIONNAME* **)**

Description

This macro should be put as the first statement in every function in the application source code. The argument to this macro is the function name. Note that this argument is not a character string. It is used by the macro **SafDMsg()** to print the name of the function.

The category format directive '%f' is used to print the function name. For more information refer to **SafDAddCategory()**.

Parameters

FUNCTIONAME The name of the function.

Example

```
void MyClass::MyFunction()
{
   SafDFunction( MyFunction );

   …
}
```

See Also: SafDMsg, SafDAddCategory

SafDImplementCategory

SafDImplementCategory(*CATEGORY* **)**

Description

This macro defines a category. It must be put in a *C* file.

The argument *CATEGORY* is the identification of the category. The same identification must be passed as argument to the macros **SafDDeclareCategory()** and **SafDAddCategory()**.

Note that this macro must not be put inside a **SafDImplementReportBegin()** and **SafDImplementReportEnd()** block.

Parameters

CATEGORY Name of the category.

Example

For a complete example refer to the description of the macro **SafDDeclareReport()**.

See Also: SafDDeclareCategory, SafDAddCategory

SafDImplementClass

SafDImplementClass(*CLASS* **)**

Description

This macro is used together with the macro **SafDDeclareClass** to define the class name used by the macro **SafDMsg()**.

The category format directive '%C' is used to print the class name. For more information refer to **SafDAddCategory()**.

This macro must be added outside the class definition.

Parameters

CLASS The name of the class.

Example

For a complete example refer to the description of the macro **SafDDeclareClass**.

See Also: SafDDeclareClass

SafDImplementContext

`SafDImplementContext(` *`CONTEXT`* **`)`**

Description

This macro defines a context. It must be put in a *C* file.

The argument *CONTEXT* is the identification of the context. The same identification must be passed as argument to the macros **SafDDeclareContext()** and **SafDAddContext()**.

Note that this macro must not be put inside a **SafDImplementReportBegin()** and **SafDImplementReportEnd()** block.

Parameters

CONTEXT Identification of the context.

Example

For a complete example refer to the description of the macro **SafDDeclareReport()**.

See Also: SafDAddContext, SafDDeclareContext

SafDImplementReportBegin

`SafDImplementReportBegin(` *`NAME`* **`)`**

Description

This macro marks the beginning of a **SafReport** definition block. Together with the macro **SafDImplementReportEnd()** it is used to mark a block where **SafReport** objects can be defined using the macros **SafDAddCategory()**, **SafDAddContext()** and **SafDChangeCategoryFormat()**. Note that all definitions of **SafReport** objects must be inside such block.

This macro must be put in a *C* file. The parameter *NAME* used must be equal to the argument used for the macros **SafDDeclareReport()** and **SafDImplementReportEnd()**.

Parameters

NAME Unique name used to mark the beginning of **SafReport** definition block.

Example

For a complete example refer to the description of the macro **SafDDeclareReport()**.

See Also: SafDImplementReportEnd, SafDDeclareReport

SafDImplementReportEnd

SafDImplementReportEnd(*NAME* **)**

Description

This macro marks the end of a **SafReport** definition block. It must be used together with **SafDImplementReportBegin()**. For more information refer to the description for the macro **SafDImplementReportBegin()**.

This macro must be put in a *C* file. The parameter *NAME* used must be equal to the argument used for the macros **SafDDeclareReport()** and **SafDImplementReportBegin()**.

Parameters

NAME Unique name used to mark the end of **SafReport** definition block.

Example

For a complete example refer to the description of the macro **SafDDeclareReport()**.

See Also: SafDImplementReportBegin, SafDDeclareReport

SafDMsg

SafDMsg(*CATEGORY, CONTEXT, ARGUMENTS* **)**

Description

This macro prints a message to an output device and to a file. It is printed only if the corresponding category and context entry in the **SafReport** matrix is enabled.

The parameter *ARGUMENTS* must be surrounded by parenthesis. *ARGUMENTS* can be plain characters or argument directives. It uses the same format as *printf()*. For more information about this format refer to *printf()* documentation.

The category format directive '%t' is used to print the parameter *ARGUMENT*. For more information refer to **SafDAddCategory()**.

Parameters

CATEGORY Name of the category.

CONTEXT Name of the context.

ARGUMENTS Format and arguments surrounded by parenthesis.

Example

This example demonstrates the use of the macro **SafDMsg()** to print messages to an output device and file. The method *SubscribeTo(SbTSTI id)* and the member variables are not of interest in this example and are therefore skipped.

```
void MyClass::SubscribeTo()
{
   SafDFunction( SubScribeTo );

   SafDMsg( SAF CAT INFO, APP SUBS,
            ( "Subscribing to variables\n" ) );

   int numFailed = 0;
   for ( int j = 0; j < mNumVariables; j++ )
   {
      if ( SubscribeTo( mVariable[ j ] ) == -1 )
      {
         // Error
         SafDMsg( SAF CAT ERROR, APP SUBS,
                  ( "Failed to subscribe to %s\n",
                     SbName( mVariable[ j ] ) );
         numFailed++;
      }
   }

   if ( numFailed > 0 )
      SafDMsg( SAF CAT DEBUG, APP SUBS,
               ( "Failed to subscribe to %d variables\n",
                  numFailed ) );
}
```

See Also: SafDFunction, SbName

SafReportExternalInterface

The class **SafReportExternalInterface** is used to access categories and contexts in a remote *SAF* compliant task. It can use this access to change the amount of information printed to an output device by the remote task.

Methods for enabling and disabling categories and contexts are part of the interface.

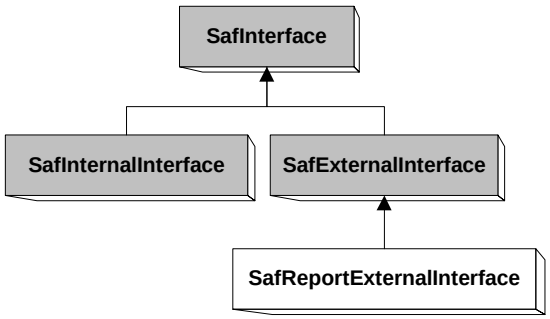
This class is derived from **SafExternalInterface** but it should be further sub-classed to overload the virtual methods **onConnect()** and **onDisconnect()**. Note that the method **get()** can not be used to retrieve the contexts and categories from the remote process before the **onConnect()** method is invoked.

Even if this class derives from **SafExternalInterface** no internal interface needs to be created in the remote task. It will automatically be handled by *SAF*.

The information flow a **SafReportExternalInterface** instance can change in a remote task depends on a number of factors. Most important is where and how frequent the macro **SafDMsg()** is used in the remote task. Another factor is the number of categories and contexts defined in the remote task.

In a *SAF* compliant application never use **printf()**, **cout**, **cerr** or any other function that prints information directly to a screen. If possible always use the macro **SafDMsg()**. Refer to **SafReport** for more information about this macro.

The structs **SafContext** and **SafCategory** are used in **SafReportExternalInterface**. For more information about these definitions refer to the description for these structs.



Class Methods

Construction/Destruction	
SafReportExternalInterface	Constructs **SafReportExternalInterface** object
~SafReportExternalInterface	Destroys **SafReportExternalInterface** object

Operators	
setCategoryOutput	Enables or disables a category.
setContextOutput	Enables or disables a context.
setOutput	Enable or disable a category/context pair.

Get Methods	
categories	Returns a pointer to the categories.

contexts	Returns a pointer to the contexts.
get	Get all the categories and contexts in the remote process.
numCategories	Number of categories fetched.
numContexts	Number of contexts fetched.
output	Returns status of the categories and contexts.

Member Functions

SafReportExternalInterface::categories

```
SafCategory* categories() const
```

Description

This method returns all the categories defined in the remote process. The return value is a pointer to an array of the categories. The data type returned is **SafCategory**. For more information about this data type refer to the **SafCategory** description.

Note that no categories are available before calling the method **get()**.

Access

This function is public.

Return Value

Returns a pointer to all the categories defined in the remote process.

Example

See example for the method **output()**.

See Also: **SafReportExternalInterface::numCategories,**
SafReportExternalInterface::get,
SafReportExternalInterface::contexts,
SafReportExternalInterface::output

SafReportExternalInterface::contexts

```
SafContext* contexts() const
```

Description

This method returns all the contexts defined in the remote process. The return value is a pointer to an array of all the contexts. The data type returned is **SafContext**. For more information about this data type refer to the description of **SafContext**.

Note that no contexts are available before calling the method **get()**.

Access

This function is public.

Return Value

Returns a pointer to all the contexts defined in the remote process.

Example

See example for the method **output()**.

See Also: **SafReportExternalInterface::numContexts,**
SafReportExternalInterface::get,
SafReportExternalInterface::categories,
SafReportExternalInterface::output

SafReportExternalInterface::get

```
SafStatus get()
```

Description

This method gets all information about the contexts and the categories in the remote process. Note that this method must be called before any of the other methods in the class.

Calling this method will remove old information stored in the object and get new information. This method should be called after calling one of the **set…()** methods. Never store references to internal structures in **SafReportExternalInterface**, i.e. return values from **output()**, **categories()** and **contexts()**. Memory areas will be changed when calling this method.

Do not call this method before the **onConnect()** method is invoked. For more information about **onConnect()** refer to the description of **SafInterface**.

Access

This function is public.

Return Value

Returns *Saf_OK* if the information was successfully retrieved from the remote process, otherwise one of the following error codes will be returned:

- *Saf_SbCreateFailed* Failed to create *SWBus* variables. Internal error.
- *Saf_SbCallFailed* Failed to call *SWBus* function in the remote process.

Example

See example for the method **output()**.

See Also: **SafReportExternalInterface::categories,**
SafReportExternalInterface::contexts,
SafReportExternalInterface::output

SafReportExternalInterface::numCategories

```
int32 numCategories() const
```

Description

This method returns the number of categories retrieved from the remote process.

Note that this information is not available before calling the method **get()**.

Access

This function is public.

Return Value

Returns number of retrieved categories.

Example

See example for the method **output()**.

See Also: SafReportExternalInterface::categories

SafReportExternalInterface::numContexts

```
int32 numContexts() const
```

Description

This method returns the number of contexts retrieved from the remote process.

Note that this information is not available before calling the method **get()**.

Access

This function is public.

Return Value

Returns number of retrieved contexts.

Example

See example for the method **output()**.

See Also: SafReportExternalInterface::contexts

SafReportExternalInterface::output

```
SafTBool* output() const
```

Description

This method returns the status of all categories and contexts retrieved from the remote process. The return value is an array of type **SafTBool**. If a category/context pair is enabled the entry in the array is *SafTTrue*, otherwise *SafTFalse*. This array can be regarded as a matrix. To get status for a given category/context pair the following formula can be used.

index = context * **numCategories()** + category

Note that the order of the category array and context array do matter when getting the information from the methods **categories()** and **contexts()**. The order of these arrays matches the array returned from **output()**.

This information is not available before calling the method **get()**.

Access

This function is public.

Return Value

Returns a pointer to the output status array.

Example

This comprehensive example demonstrates how to implement toggle functionality. The name of the context and category are passed as arguments to the **toggle()** method.

```
SafTBool ReportEI::toggle( const char* category,
                           const char* context )
{
   // Find id of the category and the context
   int32 idCat = findCategory( category );
   int32 idCon = findContext(  context  );

   if ( idCat == -1 || idCon == -1 )
      return SafTFalse;

   // Toggle state
   SafTBool* arrOut = output();
   if ( arrOut[ idCon * numCategories() + idCat ] )
      setOutput( category, context, SafTFalse );
   else
      setOutput( category, context, SafTTrue );

   // Get updated information from the remote process
   get();

   return SafTTrue;
}

int32 ReportII::findCategory( const char* category )
{
   SafCategory* arrCat = categories();
   int32        numCat = numCategories();

   for ( int32 j = 0; j < numCat; j++ )
      if ( strcmp( arrCat[j].mString, category ) == 0 )
         return j;

   return -1;   // Not found
}

int32 ReportII::findContext( const char* context )
{
   SafContext* arrCon = contexts();
   int32       numCon = numContexts();

   for ( int32 j = 0; j < numCat; j++ )
      if ( strcmp( arrCon[j].mString, context ) == 0 )
         return j;

   return -1;   // Not found
}
```

See Also: **SafReportExternalInterface::get,**
SafReportExternalInterface::contexts,
SafReportExternalInterface::numContexts
SafReportExternalInterface::categories,
SafReportExternalInterface::numCategories

SafReportExternalInterface::SafReportExternalInterface

`SafReportExternalInterface( SafConnection*` *`connection`* **`)`**

Description

This method is the constructor for the class **SafReportExternalInterface**. It initialises the object.

Access

This constructor is public.

Parameters

connection Pointer to connection object.

See Also: SafReportExternalInterface::~SafReportExternalInterface

SafReportExternalInterface::setCategoryOutput

`SafStatus setCategoryOutput( const char*` *`category`*`,` **`SafTBool`** *`enable`* **`)`**

Description

This method enables or disables all the contexts for the supplied category in the remote task.

Access

This function is public.

Parameters

category Category to enable or disable.

enable *SafTTrue* if the category should be enabled, otherwise *SafTFalse*.

Return Value

Returns *Saf_OK* on success. If the operation failed one of the following error codes are returned:

- Saf_SbCreateFailed Failed to create input or output parameter. Internal error.
- *Saf_SbCallFailed* Failed to call the remote *SWBus* function.

See Also: SafReportExternalInterface::setContextOutput, SafReportExternalInterface::setOutput

SafReportExternalInterface::setContextOutput

`SafStatus setContextOutput( const char*` *`context`*`,` **`SafTBool`** *`enable`* **`)`**

Description

This method enables or disables all the categories for the supplied context in the remote task.

Access

This function is public.

Parameters

context	Context to enable or disable.
enable	*SafTTrue* if the context should be enabled, otherwise *SafTFalse*.

Return Value

Returns *Saf_OK* on success. If the operation failed one of the following error codes are returned:

- Saf_SbCreateFailed — Failed to create input or output parameter. Internal error.
- *Saf_SbCallFailed* — Failed to call the remote *SWBus* function

See Also: **SafReportExternalInterface::setCategoryOutput, SafReportExternalInterface::setOutput**

SafReportExternalInterface::setOutput

```
SafStatus setOutput( const char* category,
                     const char* context,
                     SafTBool    enable )
```

Description

This method enables or disables a specified category/context pair in the remote task.

Access

This function is public.

Parameters

category	Category to enable or disable.
context	Context to enable or disable.
enable	*SafTTrue* if the category/context pair should be enabled, otherwise *SafTFalse*.

Return Value

Returns *Saf_OK* on success. If the operation failed one of the following error codes are returned:

- Saf_SbCreateFailed — Failed to create input or output parameter. Internal error.
- *Saf_SbCallFailed* — Failed to call the remote *SWBus* function

Example

See example for the method **output()**.

See Also: **SafReportExternalInterface::setCategoryOutput, SafReportExternalInterface::setContextOutput**

SafReportExternalInterface::~SafReportExternalInterface

```
~SafReportExternalInterface()
```

Description

This method is the destructor for the class **SafReportExternalInterface**. All memory allocated by methods in this class are destroyed in this destructor.

Access

This destructor is public.

See Also: **SafReportExternalInterface::SafReportExternalInterface**

SafStatus

SafStatus is a data type that is used as a return type from most of the methods in *SAF*.

All **SafStatus** objects contain a unique identifier and a description. If the guidelines are followed when adding **SafStatus** objects it is guaranteed that these identifications are unique.

If trying to add a **SafStatus** object with an identification that is already used an error message will be printed and the program will terminate. This check is made when the program starts. It will not terminate at a later stage. The framework guarantees that all **SafStatus** objects are unique when running an application.

A number of **SafStatus** objects are defined in and used by *SAF*. These are defined in the header file *SafErrorCodes.h*.

Applications may add new **SafStatus** objects. *SAF* offers a set of macros helping users to add these objects. A header and a *C* file are needed. All **SafStatus** objects must be declared in header files and implemented in *C* files.

User defined **SafStatus** objects should not use identification values with a value less than 1000. These values are reserved for use by *SAF*.

Macros

Declaration	
SafDDeclareStatus	Ensures that **SafStatus** objects are initialised.
SafDStatus	Declares a **SafStatus** object.

Definition	
SafDImplementStatusBegin	Marks the beginning of **SafStatus** definitions.
SafDImplementStatusEnd	Marks the end of **SafStatus** definitions.
SafDStatusAdd	Defines a **SafStatus** object.

Global Functions

SafFStatusString	Returns the description of a **SafStatus** object.

Macros

SafDDeclareStatus

```
SafDDeclareStatus( NAME )
```

Description

This macro must be used in all header files defining **SafStatus**. It adds functionality for ensuring that all objects are initialised before any constructors for global variables are called and the main entry point in the application is entered.

Note that the name passed as argument to this macro must be unique. The same name must be used in the macros **SafDImplementStatusBegin()** and **SafDImplementStatusEnd()**.

Parameters

NAME *Unique name used to identify the **SafStatus** definition section in the C file.*

Example

For a complete example refer to the description of the macro **SafDStatus()**.

See Also: SafDimplementStatusBegin, SafDImplementStatusEnd

SafDImplementStatusBegin

`SafDImplementStatusBegin(` *NAME* **`)`**

Description

This macro together with **SafDImplementStatusEnd()** is used to mark a block where **SafStatus** objects can be defined using the macro **SafDStatusAdd()**. Note that all definitions of **SafStatus** objects must be inside a block marked with **SafDImplementStatusBegin()** and ended with **SafDImplementStatusEnd()**.

This macro must be put in a *C* file. The parameter *NAME* used must be equal to the arguments used for the macros **SafDDeclareStatus()** and **SafDImplementStatusEnd()**.

Parameters

NAME *Unique name used to mark the beginning of a **SafStatus** definition section.*

Example

For a complete example refer to the description of the macro **SafDStatus()**.

See Also: SafDImplementStatusEnd, SafDDeclareStatus

SafDImplementStatusEnd

`SafDImplementStatusEnd(` *NAME* **`)`**

Description

This macro together with **SafDImplementStatusBegin()** is used to mark a block where **SafStatus** objects can be defined using the macro **SafDStatusAdd()**.

For more information about this macro refer to the description for the macro **SafDImplementStatusBegin()**.

This macro must be put in a *C* file. The parameter *NAME* used must be equal to the arguments used for the macros **SafDDeclareStatus()** and **SafDImplementStatusBegin()**.

Parameters

NAME *Unique name used to mark the end of a **SafStatus** definition section.*

Example

For a complete example refer to the description of the macro **SafDStatus()**.

See Also: SafDImplementStatusBegin, SafDDeclareStatus

SafDStatus

SafDStatus(*SAFNAME, STATUSID* **)**

Description

This macro declares a **SafStatus** object. It is initialised with a name and a value. The value must be unique. Following the guidelines for implementing **SafStatus** objects will guarantee unique values.

All **SafStatus** declarations must be put in header files.

Parameters

SAFNAME *Name of the **SafStatus** object.*

STATUSID *The **SafStatus** unique identifier.*

Example

This is a complete example showing how to implement **SafStatus** objects. The first part declares two **SafStatus** objects in a header file. The macros described in this section are used in this example.

```
/* Filename AppSafStatus.h */

#ifndef   APP SAFSTATUS
#define __APP_SAFSTATUS__

#include <Saf.h>


#define APPCODE 1000


SafDStatus( APP OK,    APPCODE + 0 );
SafDStatus( APP ERROR, APPCODE + 1 );


SafDDeclareStatus(   APP   )


#endif /* __APP_SAFSTATUS__ */
```

The implementation continues below in the *C* file. The **SafStatus** objects are defined inside the **SafDImplementStatusBegin()**, **SafDImplementStatusEnd()** block. Note the use of *__APP__* as argument to the macros. This is the same name used in the header file.

```
/* Filename: AppSafStatus.cxx */

#include "AppSafStatus.h"

SafDImplementStatusBegin(   APP   );

SafDAdd( APP OK,    "The function returned OK" );
SafDAdd( APP ERROR, "The function failed"      );

SafDImplementStatusEnd( __APP__ );
```

See Also: **SafDImplementStatusBegin, SafDImplementStatusEnd, SafDStatusAdd**

SafDStatusAdd

SafDStatusAdd(*SAFNAME, DESCRIPTION* **)**

Description

This macro defines a **SafStatus** object. The arguments to this macro are the object name and a description. This macro must be put inside a block marked with **SafDImplementStatusBegin()** and **SafDImplementStatusEnd()**.

Note that all objects declared in the header file using **SafDStatus()** must be defined using **SafDStatusAdd()**.

Parameters

SAFNAME *Name of the **SafStatus** object.*

DESCRIPTION *Description of the **SafStatus** object.*

Example

For a complete example refer to the description of the macro **SafDStatus()**.

See Also: **SafDImplementStatusBegin, SafDImplementStatusEnd, SafDStatus**

Global Functions

SafFStatusString

const char* SafFStatusString(SafStatus *id* **)**

Description

This global function returns a pointer to the description string associated with the **SafStatus** object passed as argument. If this object is not found in *SAF* NULL is returned.

Parameters

id **SafStatus** object.

Return Value

Returns a pointer to the description string associated with the **SafStatus** object.

See Also: **SafDStatus**

SafString

A **SafString** object consists of a variable-length sequence of characters. **SafString** provides functions and operators for adding and getting character strings, together with simplified memory management, make **SafString** objects easier to use than ordinary character arrays.

SafString is based on the char data type.

SafString does not have a base class.

Class Methods

Construction/Destruction	
SafString	Constructs **SafString** object
~SafString	Destroys **SafString** object

Operators	
operator ()	Returns a pointer to the character string.
operator =	Assigns a new value to the **SafString** object.
operator ==()	Checks if two strings are equal.
operator !=()	Checks if two strings are unequal.

Member Functions

SafString::SafString

`SafString()`

`SafString( const char*` *`string`* **`)`**

`SafString( const SafString&` *`string`* **`)`**

Description

These methods are constructors for the class **SafString**.

The constructor **SafString()** initialises the object with an empty string.

The constructor **SafString(const char*** *string* **)** assigns the string passed as parameter to the object.

The constructor **SafString(const SafString&** *string* **)** assigns the character string in the **SafString** object passed as parameter to the object.

Access

These constructors are public.

Parameters

string A character string or **SafString** object to assign to the object.

See Also: SafString::~SafString, SafString::operator=

SafString::~SafString

```
~SafString()
```

Description

This method is the destructor for the class **SafString**. Removes memory allocated by the object.

Access

This destructor is public.

See Also: SafString::SafString, SafString::operator=

SafString::operator()

```
char* SafString::operator() const
```

Description

This method returns a pointer to the character string. If the character string has not been assigned, the value NULL is returned.

Access

This operator is public.

Return Value

Returns a pointer to the object's character string.

See Also: SafString::SafString, SafString::operator=

SafString::operator=

```
char* operator=( const char* s )
char* operator=( const SafString& ss )
```

Description

This operator assigns the character string *s* or the **SafString** object *ss* passed as parameter to the **SafString** object.

Access

This operator is public.

Parameters

s The character string to assign to the object.

ss The **SafString** object to assign to the object.

Return Value

Returns the string assigned to the object.

See Also: **SafString::SafString**

SafString::operator==

```
SafTBool operator==( const char* s )
SafTBool operator==( const SafString& ss )
```

Description

This operator checks if the string *s* or the **SafString** object *ss* is equal to the object's string.

Access

This operator is public.

Parameters

s	The character string to check.
ss	The **SafString** object to check.

Return Value

Returns *SafTTrue* if the string passed as argument or the **SafString** object is equal to the object's string, otherwise *SafTFalse*.

See Also: **SafString::operator!=**

SafString::operator!=

```
SafTBool operator!=( const char* s )
SafTBool operator!=( const SafString& ss )
```

Description

This operator checks if the string *s* or the **SafString** object *ss* is unequal to the object's string.

Access

This operator is public.

Parameters

s	The character string to check.
ss	The **SafString** object to check.

Return Value

Returns *SafTFalse* if the string passed as argument or the **SafString** object is equal to the object's string, otherwise *SafTTrue*.

See Also: **SafString::operator==**

SafTask

The class **SafTask** is the *SAF* representation of an *SWBus* process.

A task has a name that is used when connecting to **control**. This name is registered with **control**.

In addition a task also has a concept called a task class not found in other *SWBus* applications. The task class is used to group *SAF* processes. Instances of the same program uses different task names but have equal task classes. *SAF* programs can use this information to get for instance all *SAF* processes of a specified task class. A **SafConnection** object connected to a remote process knows about the task class of the process. This information is available in the **SafConnection** method **taskClass()**. This method returns the task class of the remote process.

For example a server program accessing information in a database may have task class “DBSERVER”, while the client program accessing information in the server program may have task class “DBCLIENT”. Several DBCLIENT programs can be started accessing information stored on the server, each with different task names but equal task class.

SafTask must be further sub-classed to enable overloading of methods. The class **SafTask** is an abstract class and can not be instantiated directly in the application. The following pure virtual methods must be overloaded in the sub-class of **SafTask**:

- **createConnection**
- **destroyConnection**

It the task is just a client for other *SAF* applications the pure virtual methods may be empty. However, it is a good practice to implement functionality for creating and destroying connection objects in these methods. If for instance a client program is developed for turning debug or flow messages on or off in processes while they are running connection objects must be created in all *SAF* applications.

Note that only one instance of a **SafTask** sub-class can be present in a *SAF* application.

The base class contains methods for registration and un-registration of the *SWBus* process with **control**. It also offers functionality for entering and terminating the main loop in the *SAF* application.

The virtual method **onVariableUpdate()** is automatically called when an external process has updated *SWBus* variables using **SbFlush()**. This method should be overloaded in the sub-class of **SafTask**.

Normally a **SafTask** sub-class instance is instantiated in the main entry function in the *SAF* application. Then the task is registered using **registerTask()**. If the application is a client, connection objects and external interfaces are created before entering the main loop.

The main loop in a *SAF* application is normally the method **enterMainLoop()**. This method calls the *SWBus* function **SbLoop()** which processes messages until the process terminates. The exceptions are GUI programs developed using for instance *Motif*, *X-Windows*, *Windows* or applications using their own main loop.

The **SafTask** object maintains a list of connection objects within the *SAF* application. Each **SafConnection** object created in the process is registered in the task object.

Class Methods

Construction/Destruction	
SafTask	Constructs **SafTask** object.

~SafTask	Destroys **SafTask** object.

Overridable Methods

establish	Called when the task is registered.
initSWBus	Called to initialise the *SWBus*.
onIncompatibleVersions	Called when the task trying to establish a connection to have an incompatible *SWBus* version.
onConnectionTimeout	Called when the task is unable to connect to the remote task.
onVariableUpdate	Called when a remote task updates some variables.

Pure Overridable Methods

createConnection	Should create a connection object.
destroyConnection	Should destroy the connection object passed as argument.

Operations

registerTask	Register the task with **control**.
taskName	Sets the name of the task.
unregister	Removes the task from **control**.

Get Methods

findConnection	Returns a pointer to a specific **SafConnection** object.
isEstablished	Returns true if the established method has been invoked.
isInMainLoop	Returns true if the program is in the main loop.
isRegistered	Returns true if the program is registered.
masterHost	Returns the name of the host running **control**.
taskClass	Returns the name of the task class.
taskName	Returns the name of the task.
theTask	Returns a pointer to the **SafTask** instance.

SWBus Methods

enterMainLoop	Enters the main loop in the *SWBus*.
terminateMainLoop	Terminates the main loop in the *SWBus*.

Member Functions

SafTask::createConnection

```
virtual SafConnection* createConnection( const char* taskClass,
                                         const char* taskName ) =0
```

Description

This is a pure virtual method that must be overloaded in sub-classes of **SafTask**. It is automatically invoked by *SAF* when a remote task is trying to establish a connection to the task.

This method is invoked when a remote task is calling the **SafConnection** method **initiateConnection()**. A message is sent from the remote task to this task requesting for a **SafConnection** object. This virtual method should create a **SafConnection** sub-class object passing the *taskName* as argument to the **SafConnection** constructor.

If the task is just a pure client not accepting any other *SAF* processes from connecting, NULL may be returned.

Note that a connection is not established before the virtual method **onConnect()** is called on the connection object.

Access

This is a pure virtual protected method. Must be implemented in sub-classes of **SafTask**.

Parameters

taskClass The remote task class.

taskName Name of the remote task.

Return Value

Should return a pointer to the created **SafConnection** sub-class instance or NULL if the connection is not accepted.

Example

This example demonstrates how a **createConnection()** method may look like. Only tasks of type “OKTASK” are accepted. The class *Task* is a sub-class of **SafTask** and *Connection* is derived from **SafConnection**.

```
SafConnection* Task::createConnection( const char* taskClass,
                                       const char* taskName )
{
   /* Only accept tasks of type OKTASK */
   if ( strcmp( taskClass, "OKTASK" ) == 0 )
      return new Connection( taskName );

   return NULL; /* Rejected */
}
```

See Also: **SafTask::destroyConnection, SafTask::SafTask, SafConnection**

SafTask::destroyConnection

virtual void destroyConnection(SafConnection* *connection* **) =0**

Description

This is a pure virtual method that must be overloaded in sub-classes of **SafTask**. It should delete the connection object passed as argument to the method.

All connection objects created in the function **createConnection()** should be deleted by this method, unless some external interface objects are kept in the connection object that would be used later. In such cases the connection object should not be deleted since it would delete all the external interface objects as well.

Access

This is a pure virtual protected method. Must be implemented in sub-classes of **SafTask**.

Parameters

connection Pointer to the connection object to delete.

Example

This example shows how a typical **destroyConnection()** method may look like.

```
void Task::destroyConnection( SafConnection* connection )
{
   delete connection;
}
```

See Also: SafTask::createConnection, SafConnection

SafTask::enterMainLoop

void enterMainLoop()

Description

By calling this method the *SAF* application enters the main loop in the *SWBus*. The *SWBus* function **SbLoop()** will process messages until it is terminated by calling the method **terminateMainLoop()**.

Access

This function is public.

See Also: SafTask::terminateMainLoop, SbLoop

SafTask::establish

virtual SafStatus establish()

Description

This virtual method is activated by *SAF* when the task calls the method **registerTask()**. The **SafTask** should be further sub-classed and this method may be overloaded.

Recommended functionality put into this method would be creation of global *SWBus* classes, variables and functions. In this context global *SWBus* objects means objects that are not related to a **SafConnection** or **SafInterface** instance.

We normally recommend that creation of *SWBus* objects are put into the establish method of **SafInterface** or **SafConnection** sub-classes, but in some situations it may be useful to create *SWBus* objects in the **establish()** method of the task. Declaration of a remote *SWBus* class used by non-*SAF* compliant applications for connecting to the process would be an example of functionality that should be put into this method.

The default implementation of this method returns *Saf_OK*.

Access

This is a virtual protected method.

Return Value

Must return *Saf_OK* if the operation completed successfully, or an error code if the method failed.

Example

This example demonstrates the use of the **establish()** method to declare a remote class used by non-*SAF* applications when connecting to this process. The methods *~notify* and *~update* are added to the remote object.

```
SafStatus Task::establish()
{
    SbTSTI rC = SbSub( SbCCRemote, "NonSAFRemote", SbCNull );

    /* Add notify and update methods */
    SbTSTI nF = SbDeclare( 0, SbCCType,            SbCNull,
                           sb notify, SbCLCC );
    SbTSTI uF = SbDeclare( 0, SbCCUpdateParameter, SbCNull,
                           sb update, SbCLCC );

    SbAdd( rC, nF, "~notify", NULL );
    SbAdd( rC, uF, "~update", NULL );

    return Saf OK;
}

SbTSTI sb_notify( SbTSTI, SbTSTI, SbTSTI in, SbTSTI out )
{
   /* Do whatever you have to do when the process connects */
   return out;
}

SbTSTI sb_update( SbTSTI, SbTSTI, SbTSTI in, SbTSTI out )
{
   /* Variables updated in the remote non-SAF application */
   return out;
}
```

See Also: SafTask::registerTask, SafInterface, SafConnection

SafTask::findConnection

SafConnection* findConnection(const char* *taskName* **) const**

Description

This method searches for the connection object identified by *taskName* among all the **SafConnection** objects stored in the task.

Access

This function is public.

Parameters

taskName The **SafConnection** object to search for.

Return Value

Returns a pointer to a **SafConnection** object if found, otherwise NULL.

See Also: SafConnection

SafTask::initSWBus

```
virtual SbTSTI initSWBus( const char* name,
                          const char* host,
                          const char* port )
```

Description

This virtual method must initialise the *SWBus* using **SbInit()**, **SbWinInit()** or **SbXtInit()**. It should only be necessary to overload this method when developing applications with a Motif or Windows GUI interface.

The default implementation of this method uses **SbInit()**.

Access

This is a virtual protected method.

Parameters

name The name of the process registered with **control**.

host Host where **control** is running.

port This parameter is ignored. Always NULL.

Return Value

Must return *SbCOk* if the initialisation was successful, otherwise *SbCError*.

Example

This example demonstrates how to initialise a *SAF* application when it is developed using *MFC*[1]on Windows.

```
SbTSTI Task::initSWBus( const char* name, const char* host,
                        const char* port )
{
   return SbWinInit( ::AfxGetInstanceHandle(),
                     name, host, port );
}
```

See Also: SafTask::registerTask, SbInit, SbWinInit, SbXtInit

SafTask::isEstablished

```
SafTBool isEstablished() const
```

Description

This method verifies if the task has been properly established and registered.

[1] MFC – Microsoft Foundation Classes

Access

This function is public.

Return Value

Returns *SafTTrue* if the task has been properly established, otherwise *SafTFalse*.

See Also: SafTask::registerTask

SafTask::isInMainLoop

```
SafTBool isInMainLoop() const
```

Description

This method verifies if the task is within the main loop of the *SWBus*.

If the application uses another main loop, for instance Motif or Windows main loop, **isInMainLoop()** will always return *SafTFalse*.

Access

This function is public.

Return Value

Returns *SafTTrue* if the task is within the main loop, otherwise *SafTFalse*.

See Also: SafTask::enterMainLoop

SafTask::isRegistered

```
static SafTBool isRegistered()
```

Description

This static method verifies if a task is registered for this process or not. A task is registered as long as it is in current scope or it is allocated on the heap.

Access

This is a static public function.

Return Value

Returns *SafTTrue* if a task is registered, otherwise *SafTFalse*.

See Also: SafTask::SafTask

SafTask::masterHost

```
const char* masterHost() const
```

Description

This method returns the name of the host where **control** is running.

Access

This function is public.

Return Value

Returns the name of the host where **control** is running.

See Also: **SafTask::registerTask, SafTask::initSWBus**

SafTask::onConnectionTimeout

virtual SafStatus onConnectionTimeout(const char* *taskName* **)**

Description

This virtual method should be overloaded in sub-classes of **SafTask**. This method is automatically invoked by *SAF* when the process has given up establishing a connection to a remote process, because the period specified by the environment variable *BUSCONNECT* has elapsed.

The default implementation of this method prints an error message and returns *Saf_OK*.

Access

This is a protected virtual method.

Parameters

taskName The name of the process given up establish a connection to.

Return Value

The default implementation returns *Saf_OK*. The overloaded method in sub-classes should return *Saf_OK* on success, otherwise an error status.

Example

This example demonstrates how a **onConnectionTimeout()** method may be implemented in a *SAF* compliant application. The class *MyTask* is a sub-class of **SafTask**.

```
SafStatus MyTask::onConnectionTimeout( const char* taskName )
{
   cout << "Timeout when trying to establish a connection to task "
        << taskName << endl;

   return Saf OK;
}
```

See Also: **SafConnection::onConnect, SafConnection::initiateConnection, SafConnection::onConnectionTimeout**

SafTask::onIcompatibleVersions

vitual SafStatus onIncompatibleVersions(const char* *taskName* **)**

Description

This virtual method is automatically invoked by *SAF* if the *SWBus* version used by the remote task trying to establish a connection to where incompatible with the *SWBus* version of the local task.

This method may be overloaded in sub-classes of **SafTask**.

The default implementation of this method prints an error message and returns *Saf_OK*.

Access

This is a virtual protected method.

Parameters

taskName The name of the process with an incompatible *SWBus* version.

Return Value

The default implementation returns *Saf_OK*. The overloaded method in sub-classes should return *Saf_OK* on success, otherwise an error status.

Example

```
SafStatus MyTask::onIncompatibleVersions( const char* taskName )
{
   cout << "Process " << taskName()
        << " is incompatible with this process" << endl;

   return Saf OK;
}
```

See Also: SafConnection::initiateConnection, SafConnection::onIncompatibleVersions

---

SafTask::onVariableUpdate

```
virtual SafStatus onVariableUpdate( const char* taskName,
                                    int32       numVariables,
                                    SbTSTI*     variables )
```

Description

This virtual method is automatically called when a remote *SAF* application has updated some variables and the *SWBus* function **SbFlush()** has been called.

This method should be overloaded in sub-classes of **SafTask**.

The default implementation of this method returns *Saf_OK*.

Access

This is a virtual protected method.

Parameters

taskName Name of the task sending the update message.

numVariables Number of variables in the *SWBus STI* array.

variables The updated variables.

Return Value

Must returns *Saf_OK* if successful, otherwise an error code.

Example

This example redistributes all the variables put into the variable array to all connected tasks.

```
SafStatus Task::onVariableUpdate( const char* taskName,
                                  int32       numVars,
                                  SbTSTI*     vars )
{
    for ( int i = 0; i < numVars; i++ )
       SbSend( vars[i] );
    SbFlush( SbCPLocal, SbCNull );

    return Saf OK;
}
```

See Also: SbFlush, SbSend, SbLink

SafTask::registerTask

SafStatus registerTask(const char* *masterHost* **= NULL)**

Description

This method registers the task with control and creates *SWBus* objects used by *SAF*. The virtual methods **initSWBus()** and **establish()** are called from this method. Normally it should be one of the first methods to invoke after creating the sub-class instance of **SafTask**.

Access

This function is public.

Parameters

masterHost Name of the host where **control** is running. If the parameter is NULL the environment variable *CONTROLHOST* is used to get the host name. If *CONTROLHOST* is NULL local host is used.

Return Value

Returns *Saf_OK* if the registration was successful, or an error code if the operation failed. The following error codes may be returned:

- *Saf_SbIllegalName* If the task has no name.
- *Saf_SbIllegalMasterHost* If the name of the local host could not be created.
- *Saf_SbInitFailed* If the *SWBus* initialisation failed.

Example

This example demonstrates how a typical main routine in a server application may look like. The class *DbTask* is derived from **SafTask**. The task class is *"DB_SERVER_CLASS"* and the task name (registered with **control**) is *"dbserver"*.

```
DbTask::DbTask( const char* taskName ) :
   SafTask( "DB SERVER CLASS", taskName )
{
   /* Initialise the Task instance */
}


int main()
{
   DbTask task( "dbserver" );
   task.registerTask();

   task.enterMainLoop();
   return 0;
}
```

See Also: SafTask::initSWBus

SafTask::SafTask

SafTask(const char* *taskClass*, **const char*** *taskName* **= NULL)**

Description

This method is the constructor for the class **SafTask**. It initialises the **SafTask** object.

The parameter *taskName* is used when connecting to the control process. This name is registered with the **control** process. This name must be unique among all the processes

connected to a specific **control** process. If not specified at creation time the method **taskName()** must be used to supply the task name before calling **registerTask()**.

In *SAF* a concept called a task class is used that is not found in other *SWBus* application. The task class is used to group *SAF* processes. Instances of the same program uses different task names but have equal task classes. *SAF* programs can use this information to get for instance all *SAF* processes of a specific task class. The parameter *taskClass* should be unique among different types of *SAF* programs.

Access

This constructor is public.

Parameters

taskClass	Name of the task class.
taskName	Name of the task. Optional.

Example

Refer to **registerTask()** for an example.

See Also: **SafTask, SafTask::~SafTask, SafTask::taskName, SafTask::registerTask**

SafTask::taskClass

```
const char taskClass() const
```

Description

This method returns the name of the task class.

Access

This function is public.

Return Value

Returns the task class.

See Also: **SafTask::SafTask**

SafTask::taskName

```
SafStatus taskName( const char* taskName )
const char* taskName() const
```

Description

The method **taskName(const char*** *taskName* **)** supplies a new task name for the application. It must be called before the method **registerTask()** is invoked.

The method **taskName()** returns current task name.

Access

These functions are public.

Parameters

taskName	New name of the task.

Return Value

taskName(const char* *taskName* **)** returns *Saf_OK* if the name was stored, otherwise *Saf_NullArg*.

taskName() returns the name of the task.

See Also: SafTask::SafTask

SafTask::terminateMainLoop

```
void terminateMainLoop()
```

Description

This method terminates the main loop in the *SAF* application. The *SWBus* function **SbEndLoop()** is called.

Access

This function is public.

See Also: SafTask::enterMainLoop

SafTask::unregister

```
SafStatus unregister()
```

Description

This method is automatically called from the class destructor. It disconnects from all the connected tasks, releases memory occupied by the *SWBus* objects and finally leaves the *SWBus* by calling **SbExit()**.

Access

This function is public.

Return Value

Returns *Saf_OK*.

See Also: SafTask::~SafTask, SafTask::registerTask, SbExit

SafTask::~SafTask

```
virtual ~SafTask()
```

Description

This method is the destructor for the class **SafTask**. It destroys all memory occupied by the task object and calls the method **unregister()** to disconnect from all connected tasks. For more information refer to **unregister()**.

Access

This is a virtual public destructor.

See Also: SafTask::SafTask, SafTask::unregister