SC23-2681

Bull DPX/20

SOMobjects Base Toolkit
Programmer's Reference Manual

AIX

ORDER REFERENCE
86 A2 28AQ 01

Bull DPX/20

SOMobjects Base Toolkit Programmer’s Reference Manual

AIX

Software

June 1995

BULL S.A. CEDOC
Atelier de Reproduction
FRAN–231
331 Avenue Patton BP 428
49005 ANGERS CEDEX
FRANCE

ORDER REFERENCE
86 A2 28AQ 01

Suggestions and criticisms concerning the form, content, and presentation of this book are invited. A form is provided at the end of this book for this purpose.

Trademarks and Acknowledgements

About This Book

This book gives reference material for the **System Object Model** (**SOM**) of the **SOMobjects Base Toolkit.** In particular, it contains a reference page for every class, method, function, and macro provided by the SOM run-time library, the DSOM run-time library, the Interface Repository Framework, and the Event Management Framework. It also includes documentation of the utility metaclasses provided by the SOMobjects Base Toolkit, and each of their methods.

Also, the *SOMobjects Base Toolkit Quick Reference Guide* shows the syntax and purpose for each entry of the current book, plus SOM Compiler commands/flags. In addition, refer to the *SOMobjects Base Toolkit Users Guide* for introductory information.

Who Should Use This Book

This book is for the professional programmer using the SOMobjects Base Toolkit to build object-oriented class libraries or application programs that use SOM class libraries or the frameworks in the SOMobjects Base Toolkit.

This book assumes that you are an experienced programmer and that you have a general familiarity with the basic notions of object-oriented programming. Practical experience using an object-oriented programming language is helpful, but not essential.

How This Book Is Organized

At the highest level, this book is organized by framework. Within each framework, the reference pages describe the classes in alphabetical order, with the methods of each class given in alphabetical order following their corresponding class. Similarly, related functions and SOM macros are given in separate alphabetical sequences in the corresponding section. The reference page for a SOM **class** contains the following topics:

Description	A description of the class.
File Stem	The file stem for the class's IDL interface specification (.idl) file and its usage binding (.h/.xh) files.
Base Class	The class's direct base (parent) classes.
Ancestor Classes	The class's ancestor (indirect base) classes.
Metaclass	The class's metaclass.
New Methods	The names of the methods that the class introduces (grouped roughly according to purpose). Each new method is documented on a separate reference page.
Overriding Methods	The names of the methods that the class overrides from ancestor classes

The reference page for a **method** of a SOM class contains the following topics:

Purpose	The purpose of the method in brief.
Syntax	The method's C/C++ procedure prototype (which includes the method procedure's return type and the names and types of its parameters). The in/out/inout keywords associated with each of the method's parameters in the method's IDL declaration are also shown. These keywords are shown for information only; they are not actually present in the method procedure prototype.
Description	A description of the method's use.
Parameters	A description of each of the method procedure's parameters.

Return Value	A description of the method's return value.
Example	An example of using or overriding the method, if available. Although methods of SOM classes are language neutral (that is, they can be invoked from any programming language that can use SOM), the examples given here are written in C.
Original Class	The name of the class that introduces the method (the class is documented separately in this book).
Related Information	Related methods and functions (and macros, for the SOM kernel) that can be found in this book.

The reference page for a **function** has the following topics:

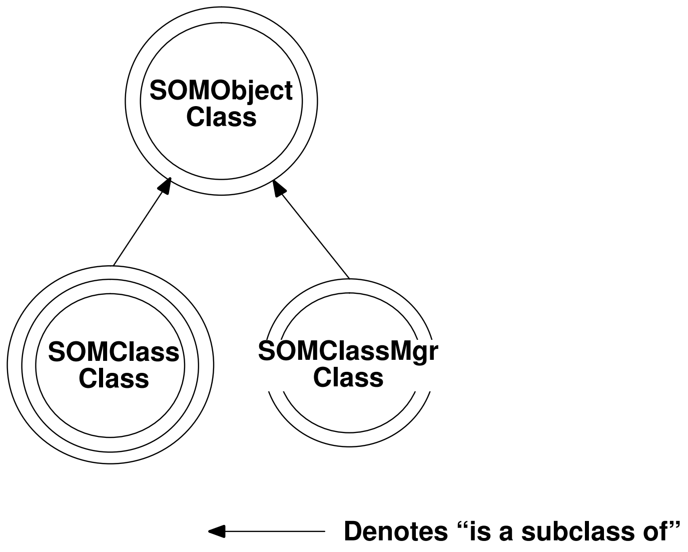
Purpose	The purpose of the function in brief.
Syntax	The function's prototype (which includes the return type and the names and types of the parameters).
Description	A description of the function's use.
Parameters	A description of each of the function's parameters.
Return Value	A description of the function's return value.
Example	An example of using the function, if available.
Related Information	Related methods and functions (and macros, for the SOM kernel) that can be found in this book.

The reference page for a **macro** has the following fields:

Purpose	The purpose of the macro in brief.
Syntax	The syntax for invoking the macro.
Description	A description of the macro's use.
Parameters	A description of each of the macro's parameters.
Expansion	A description of the macro's expansion (although the exact code expansion is not always given).
Example	An example of invoking the macro, if available.
Related Information	Related macros and functions that can be found in this book.

Contents

Chapter 1. SOM Kernel Reference



SOM Kernel Class Organization

somApply Function

Purpose

Invokes an apply stub. Apply stubs are never invoked directly by SOM users, the **somApply** function must be used instead.

Syntax

boolean somApply (
SOMObject *objPtr*,
somToken **retVal*,
somMethodDataPtr *mdPtr*,
va_list *args***);**

Description

somApply provides a single uniform interface through which it is possible to call any method procedure. The interface is based on the caller passing: the object to which the method procedure is to be applied; a return address for the method result; a *somMethodDataPtr* indicating the desired method procedure; and an ANSI standard **va_list** structure containing the method procedure arguments. Different method procedures expect different argument types and return different result types, so the purpose of **somApply** is to select an *apply stub* appropriate for the specific method involved, according to the supplied method data, and then call this apply stub. The apply stub removes the arguments from the **va_list**, calls the method procedure with these arguments, accepts the returned result, and then copies this result to the location pointed to by *retVal*.

The method procedure used by the apply stub is determined by the content of the **somMethodData** structure pointed to by *mdPtr*. The class methods **somGetMethodData** and **somGetNthMethodData** are used to load a **somMethodData** structure. These methods resolve static method procedures based on the receiving class's instance method table.

The SOM API requires that information necessary for selecting an apply stub be provided when a new method is registered with its introducing class (by way of the methods **somAddStaticMethod** or **somAddDynamicMethod**). This is required because SOM itself needs apply stubs when dispatch method resolution is used. C and C++ implementation bindings for SOM classes support this requirement, but SOM does not terminate execution if this requirement is not met by a class implementor. Thus, it is possible that there may be methods for which **somApply** cannot select an appropriate apply stub. The **somMethodData** structure for the method can be inspected before calling **somApply** to verify that the method data contains sufficient information to select an appropriate apply stub: either the *applyStub* component or the *stubInfo* component of this structure must be non-NULL. If these conditions are met, then **somApply** performs as described previously, and a TRUE value is returned; otherwise FALSE is returned.

Parameters

objPtr	A pointer to the object on which the method procedure is to be invoked.
retVal	A pointer to the memory region into which the result returned by the method procedure is to be copied. This pointer **cannot** be null (even in the case of method procedures whose returned result is void).
mdPtr	A pointer to the **somMethodData** structure that describes the method whose procedure is to be executed by the apply stub.

args A pointer to a memory region in which all of the arguments to the method procedure have been laid out in consecutive addresses, according to the protocol implemented by **va_lists**. The first entry of the **va_list must** be *objPtr*. Furthermore, all arguments on the **va_list** must appear in widened form, as defined by ANSI C. For example, **floats** must appear as **doubles**, and **chars** and **shorts** must appear as **ints**.

C++ Example

```
#include <somcls.xh>
#include <string.h>
#include <stdarg.h>
main()
{   va_list args = (va_list) SOMMalloc(4);
    va_list push = args;
    string result;
    SOMClass *scObj;
    somMethodData md;

    somEnvironmentNew(); /* Init environment */
    scObj = _SOMClass;   /* The SOMClass object */

    scObj->somGetMethodData(somIdFromString(”somGetName”), &md);
    va_arg(push, SOMClass*) = scObj;

    somApply(scObj, (somToken*)&result, &md, args);
    SOM_Assert(!strcmp(result,”SOMClass”), SOM_Fatal);
    /* result is ”SOMClass” */
}
```

Related Information

Methods: somGetMethodData, somGetNthMethodData, somGetRdStub, somAddStaticMethod, somAddDynamicMethod (somcls.idl)

Data Structures: SOMObject (**somobj.idl**), **somMethodData** (**somapi.h**), **somToken** (**somapi.h**), **somMethodPtr** (**sombtype.h**), **va_list** (**stdarg.h**)

somBeginPersistentIds Function

Purpose

Tells SOM to begin a “persistent ID interval.”

Syntax

void somBeginPersistentIds ( );

Description

The **somBeginPersistentIds** function informs the SOM ID manager that strings for any new SOM IDs that are registered will not be freed or modified. This allows the ID manager to use a pointer to the string in the unregistered ID as the master copy of the ID’s string, rather than making a copy of the string. This makes ID handling more efficient.

C Example

```
#include <som.h>
/* This is the way to create somIds efficiently */
static string id1Name = ”whoami”;
static somId somId_id1 = &id1Name;
/*
   somId_id1 will be registered the first time it is used
   in an operation that takes a somId, or it can be explicitly
   registered using somCheckId.
*/

main()
{
   somId id1, id2;
   string id2Name = ”whereami”;

   somEnvironmentNew();
   somBeginPersistentIds();
   id1 = somCheckId(somId_id1); /* registers the id as persistent
*/
   somEndPersistentIds();
   id2 = somIdFromString(id2Name); /* registers the id */

   SOM_Assert(!strcmp(”whoami”, somStringFromId(id1)),
SOM_Fatal);
   SOM_Assert(!strcmp(”whereami”, somStringFromId(id2)),
SOM_Fatal);

   id1Name = ”it does matter”; /* because it is persistent */
   id2Name = ”it doesn’t matter”; /* because it is not persistent
*/

   SOM_Assert(strcmp(”whoami”, somStringFromId(id1)), SOM_Fatal);
   /* The id1 string has changed */
   SOM_Assert(!strcmp(”whereami”, somStringFromId(id2)),
SOM_Fatal);
   /* the id2 string has not */
}
```

Related Information

Functions: somCheckId, **somRegisterId**, **somIdFromString**, **somStringFromId**, **somCompareIds**, **somTotalRegIds**, **somSetExpectedIds**, **somEndPersistentIds**, **somUniqueKey**

somBuildClass Function

Purpose

Automates the process of building a new SOM class object.

Syntax

```
void  somBuildClass (
                        unsigned long inheritVars,
                        somStaticClassInfoPtr sciPtr,
                        long majorVersion,
                        long minorVersion);
```

Description

The **somBuildClass** function accepts declarative information defining a new class that is to be built, and performs the activities required to build and register a correctly functioning class object. The C and C++ implementation bindings use this function to create class objects.

Parameters

inheritVars	A bit mask that determines inheritance from parent classes. A mask containing all ones is an appropriate default.
sciPtr	A pointer to a structure holding static class information.
majorVersion	The major version number for the class.
minorVersion	The minor version number for the class.

Example

See any **.ih** or **.xih** implementation binding file for details on construction of the required data structures.

Related Information

Data Structures: somStaticClassInfo (**somapi.h**)

somCheckId Function

Purpose

Registers a SOM ID.

Syntax

somId somCheckId (somId *id***);**

Description

The **somCheckId** function registers a SOM ID and converts it into an internal representation. The input SOM ID is returned. If the ID is already registered, this function has no effect.

Parameters

id	The **somId** to be registered.

Return Value

The registered **somId.**

Example

See the **somBeginPersistentIds** function.

Related Information

Functions: somRegisterId, **somIdFromString**, **somStringFromId**, **somCompareIds**, **somTotalRegIds**, **somSetExpectedIds**, **somBeginPersistentIds**, **somEndPersistentIds**, **somUniqueKey**

Data Structures: somId (**sombtype.h**)

somClassResolve Function

Purpose

Obtains a pointer to the procedure that implements a static method for instances of a particular SOM class.

Syntax

somMethodPtr somClassResolve (SOMClass *cls,* **somMToken** *mToken***);**

Description

The **somClassResolve** function is used to obtain a pointer to the procedure that implements the specified method for instances of the specified SOM class. The returned procedure pointer can then be used to invoke the method. The somClassResolve function is used to support “casted” method calls, in which a method is resolved with respect to a specified class rather than the class of which an object is a direct instance. The **somClassResolve** function can only be used to obtain a method procedure for a static method (a method declared in an IDL specification for a class); dynamic methods do not have method tokens.

The SOM language usage bindings for C and C++ do not support casted method calls, so this function must be used directly to achieve this functionality. Whenever using SOM method procedure pointers, it is necessary to indicate the use of system linkage to the compiler. The way this is done depends on the compiler and the system being used. However, C and C++ usage bindings provide an appropriate typedef for this purpose. The name of the typedef is based on the name of the class that introduces the method, as illustrated in the following example.

Parameters

cls	A pointer to the class object whose instance method procedure is required.
mToken	The method token for the method to be resolved. The SOM API requires that if the class “XYZ” introduces the static method “foo”, then the method token for “foo” is found in the class data structure for “XYZ” (called XYZClassData) in the structure member named “foo” (that is, at XYZClassData.foo). Method tokens can also be obtained using the **somGetMethodToken** method.

Return Value

A **somMethodPtr** pointer to the procedure that implements the specified method for the specified class of SOM object.

C++ Example

```
// SOM IDL for class A and class B
#include <somobj.idl>
module scrExample {
   interface A : SOMObject { void foo(); implementation {
                             callstyle=oidl; }; };
   interface B : A  {  implementation { foo: override; }; };
};

// Example C++ program to implement and test module scrExample
#define SOM_Module_screxample_Source
#include <scrExample.xih>
#include <stdio.h>

SOM_Scope void SOMLINK scrExample_Afoo(scrExample_A *somSelf);
{  printf(”1\n”); }

SOM_Scope void SOMLINK scrExample_Bfoo(scrExample_B *somSelf);
{ printf(”2\n”); }

main()
{
   scrExample_B  *objPtr = new scrExample_B;

   // This prints 2
   objPtr->foo();

   // This prints 1
   ((somTD_scrExample_A_foo) /* A necessary method procedure cast
*/
      somClassResolve(
          _scrExample_A, // the A class object
          scrExample_AClassData.foo) // the foo method token
      ) /* end of method procedure expression */
      (objPtr); /* method arguments */

   // This prints 2
   ((somTD_scrExample_A_foo) /* A necessary method procedure cast
*/
      somClassResolve(
          _scrExample_B, // the B class object
          scrExample_AClassData.foo) // the foo method token
      ) /* end of method procedure expression */
      (objPtr); /* method arguments */

}
```

Related Information

Functions: somResolveByName, **somParentResolve**, **somParentNumResolve**, **somResolve**

Data Structures: somMethodPtr (**sombtype.h**), **SOMClass** (**somcls.idl**), **somMToken** (**somapi.h**)

Methods: somDispatch, **somClassDispatch**, **somFindMethod, somFindMethodOk**, **somGetApplyStub**, **somGetMethodToken**

Macros: SOM_Resolve, **SOM_ResolveNoCheck**

somCompareIds Function

Purpose

Determines whether two SOM IDs represent the same string.

Syntax

int somCompareIds (somId *id1,* **somId** *id2***);**

Description

The **somCompareIds** function returns 1 if the two input IDs represent strings that are equal; otherwise, it returns 0.

Parameters

id1	The first SOM ID to be compared.
id2	The second SOM ID to be compared.

Return Value

Returns 1 if the two input IDs represent strings that are equal; otherwise, it returns 0.

C Example

```
#include <som.h>
main()
{
   somId id1, id2, id3;

   somEnvironmentNew();
   id1 = somIdFromString("this");
   id2 = somIdFromString("that");
   id3 = somIdFromString("this");

   SOM_Test(somCompareIds(id1, id3));
   SOM_Test(! somCompareIds(id1, id2));
}
```

Related Information

Functions: somCheckId, **somRegisterId**, **somIdFromString**, **somStringFromId**, **somTotalRegIds**, **somSetExpectedIds**, **somUniqueKey**, **somBeginPersistentIds**, **somEndPersistentIds**

Data Structures: somId (**sombtype.h**)

somDataResolve Function

Purpose

Accesses instance data within an object.

Syntax

somToken somDataResolve (SOMObject *obj,* **somDToken** *dToken***);**

Description

The **somDataResolve** function is used to access instance data within an object. This function is of use primarily to class implementors (rather than class clients) who are not using the SOM C or C++ language bindings.

For C or C++ programmers with access to the C or C++ implementation bindings for a class, instance data can be accessed using the <*className*>**GetData** macro (which expands to a usage of **somDataResolve**).

Parameters

obj	A pointer to the object whose instance data is required.
dToken	A data token for the required instance data. The SOM API specifies that the data token for accessing the instance data introduced by a class is found in the *instanceDataToken* component of the auxiliary class data structure for that class. The example which follows illustrates this.

Return Value

A **somToken** (that is, a pointer) that points to the data in *ob*j identified by the *dToken*.

C Example

The following C/C++ expression evaluates to the address of the instance data introduced by class “XYZ” within the object “obj”. This assumes that “obj” points to an instance of “XYZ” or a subclass of “XYZ”.

```
include <som.h>
somDataResolve(obj, XYZCClassData.instanceDataToken)
```

Related Information

Data Structures: somToken (**somapi.h**), **SOMObject** (**somobj.idl**),
somDToken (**somapi.h**)

somEndPersistentIds Function

Purpose

Tells SOM to end a “persistent ID interval.”

Syntax

void somEndPersistentIds ( );

Description

The **somEndPersistentIds** function informs the SOM ID manager that strings for any new SOM IDs that are registered might be freed or modified by the client program. Thus, the ID manager must make a copy of the strings.

Example

See the **somBeginPersistentIds** function.

Related Information

Functions: somCheckId, **somRegisterId**, **somIdFromString**, **somStringFromId**, **somCompareIds**, **somTotalRegIds**, **somSetExpectedIds**, **somUniqueKey**, **somBeginPersistentIds**

somEnvironmentEnd Function

Purpose

Provides general cleanup for applications.

Syntax

void somEnvironmentEnd ( );

Description

The **somEnvironmentEnd** function is a general cleanup function that must be called by all Windows applications before exiting. AIX and OS/2 programs may also invoke this function, but it is not required on these systems because all necessary SOM cleanup is performed by the operating system during program termination.

A convenience macro, **SOM_MainProgram,** which usually appears at the beginning of each application, adds the **somEnvironmentEnd** function to the "atexit" list. If the "atexit" mechanism does not work reliably with your compiler, or if you know that your program bypasses the normal program termination sequence, you should insert an explicit call to **somEnvironmentEnd** at the point where your main program exits. (All main programs for Windows must begin either with the **SOM_MainProgram** macro or with a call to the **somMainProgram** function.)

Related Information

Macros: SOM_MainProgram

somEnvironmentNew Function

Purpose

Initializes the SOM runtime environment.

Syntax

SOMClassMgr somEnvironmentNew ( );

Description

The **somEnvironmentNew** function creates the four primitive SOM objects (*SOMObject, SOMClass, SOMClassMg*r, and *SOMClassMgrObject*) and initializes global variables used by the SOM runtime environment. This function must be called before using any other SOM functions or methods (with the exception of **somSetExpectedIds**). If the SOM runtime environment has already been initialized, calling this function has no harmful effect.

Although this function must be called before using other SOM functions or methods, it needn't always be called explicitly, because the <*className*>**New** macros, the <*className*>**Renew** macros, the **new** operator, and the <*className*>**NewClass** procedures defined by the SOM C and C++ language bindings call **somEnvironmentNew** if needed.

Return Value

A pointer to the single class manager object active at run time. This class manager can be referred by the global variable *SOMClassMgrObject*.

Example

```
somEnvironmentNew();
```

Related Information

Functions: somExceptionId, **somExceptionValue**, **somSetException**, **somGetGlobalEnvironment**

somExceptionFree Function

Purpose

Frees the memory held by the exception structure within an **Environment** structure.

Syntax

void somExceptionFree (Environment ****ev*);**

Description

The **somExceptionFree** function frees the memory held by the exception structure within an **Environment** structure.

Parameters

ev	A pointer to the **Environment** whose exception information is to be freed.

Example

See the **somSetException** function.

Related Information

Functions: somExceptionId, **somExceptionValue**, **somSetException**, **somGetGlobalEnvironment**, **somdExceptionFree** (DSOM function)

Data Structures: Environment (**somcorba.h**)

somExceptionId Function

Purpose

Gets the name of the exception contained in an **Environment** structure.

Syntax

string somExceptionId (Environment *****ev*);**

Description

The **somExceptionId** function returns the name of the exception contained in the specified **Environment** structure.

Parameters

ev	A pointer to an **Environment** structure containing an exception.

Return Value

The **somExceptionId** function returns the name of the exception contained in the specified **Environment** structure, as a string.

Example

See the **somSetException** function.

Related Information

Functions: somExceptionValue, **somSetException**, **somGetGlobalEnvironment**, **somdExceptionFree**

Data Structures: string (**somcorba.h**), **Environment** (**somcorba.h**)

somExceptionValue Function

Purpose

Gets the value of the exception contained in an **Environment** structure.

Syntax

somToken somExceptionValue (Environment ***_ev_);**

Description

The **somExceptionValue** function returns the value of the exception contained in the specified **Environment** structure.

Parameters

ev	A pointer to an **Environment** structure containing an exception.

Return Value

The **somExceptionValue** function returns a pointer to the value of the exception contained in the specified **Environment** structure.

Example

See the **somSetException** function.

Related Information

Functions: somExceptionId, **somdExceptionFree**, **somSetException**, **somGetGlobalEnvironment**

Data Structures: somToken (**somapi.h**), **Environment** (**somcorba.h**)

somGetGlobalEnvironment Function

Purpose

Returns a pointer to the current global **Environment** structure.

Syntax

Environment *somGetGlobalEnvironment ( );

Description

The **somGetGlobalEnvironment** function returns a pointer to the current global **Environment** structure. This structure can be passed to methods that require an **(Environment *)** argument. The caller can determine if the called method has raised an exception by testing whether

```
ev->_major != NO_EXCEPTION
```

If an exception has been raised, the caller can retrieve the name and value of the exception using the **somExceptionId** and **somExceptionValue** functions.

Return Value

A pointer to the current global **Environment** structure.

Example

See the **somSetException** function.

Related Information

Functions: somExceptionId, **somdExceptionFree**, **somSetException**, **somExceptionValue**

Data Structures: Environment (**somcorba.h**)

somIdFromString Function

Purpose

Returns the SOM ID corresponding to a given text string.

Syntax

somId somIdFromString (string *aString)*;

Description

The **somIdFromString** function returns the SOM ID that corresponds to a given text string.

Ownership of the **somId** returned by **somIdFromString** passes to the caller, which has the responsibility to subsequently free the **somId** using **SOMFree**.

Parameters

aString	The string to be converted to a SOM ID.

Return Value

Returns the SOM ID corresponding to the given text string.

Example

See the **somBeginPersistentIds** function.

Related Information

Functions: somCheckId, **somRegisterId**, **somStringFromId**, **somCompareIds**, **somTotalRegIds**, **somSetExpectedIds**, **somUniqueKey**, **somBeginPersistentIds**, **somEndPersistentIds**

Data Structures: somId (**sombtype.h**), **string** (**somcorba.h**)

somIsObj Function

Purpose

Failsafe routine to determine whether a pointer references a valid SOM object.

Syntax

boolean somIsObj (somToken *memPtr***)**;

Description

The **somIsOb**j function returns 1 if its argument is a pointer to a valid SOM object, or returns 0 otherwise. The function handles address faults, and does extensive consistency checking to guarantee a correct result.

Parameters

memPtr	A **somToken** (a pointer) to be checked.

Return Value

The **somIsObj** function returns 1 if *obj* is a pointer to a valid SOM object, and 0 otherwise.

C++ Example

```
#include <stdio.h>
#include <som.xh>

void example(void *memPtr)
{
   if (!somIsObj(memPtr))
     printf(”memPtr is not a valid SOM object.\n”);
   else
     printf(”memPtr points to an object of class %s\n”,
             ((SOMObject *)memPtr)->somGetClassName());
}
```

Related Information

Data Structures: boolean (**somcorba.h**), **somToken** (**somapi.h**)

somLPrintf Function

Purpose

Prints a formatted string in the manner of the C printf function, at the specified indentation level.

Syntax

long somLPrintf (long *level,* **string** *fmt,* ...**);**

Description

The **somLPrintf** function prints a formatted string using SOMOutCharRoutine, in the same manner as the C printf function. The implementation of SOMOutCharRoutine determines the destination of the output, while the C printf function is always directed to **stdout**. (The default output destination for SOMOutCharRoutine is **stdout** also, but this can be modified by the user). The output is prefixed at the indicated level, by preceding it with 2*level spaces.

Parameters

level	The level at which output is to be placed.
fmt	The format string to be output.
varargs	The values to be substituted into the format string.

Return Value

Returns the number of characters written.

C Example

```
#include <somobj.h>
somLPrintf(5, ”The class name is %s.\n”, _somGetClassName(obj));
```

Related Information

Functions: somVprintf, **somPrefixLevel**, **somPrintf**, **SOMOutCharRoutine**

Data Structures: string (**somcorba.h**)

somMainProgram Function

Purpose

Performs SOM initialization on behalf of a new program.

Syntax

SOMClassMgr *somMainProgram ( );

Description

The **somMainProgram** function informs SOM about the beginning of a new thread of execution (called a *task* on Windows). The SOM Kernel then performs any needed initialization, including the deferred execution of the **SOMInitModule** functions found in statically-loaded class libraries. The **somMainProgram** function must appear near the beginning of all Windows main programs, and may also be used in AIX or OS/2 programs. When used, it supersedes any need to call the **somEnvironmentNew** function.

A convenience macro, **SOM_MainProgram**, which combines the execution of the **somMainProgram** function with the scheduling of the **somEnvironmentEnd** function during normal program termination, is available for C and C++ programmers.

Return Value

A pointer to the **SOMClassMgr** object.

Related Information

Functions: somEnvironmentNew, **somEnvironmentEnd**

Macros: SOM_MainProgram, **SOM_ClassLibrary**

somParentNumResolve Function

Purpose

Obtains a pointer to a procedure that implements a method, given a list of method tables.

Syntax

```
somMethodPtr somParentNumresolve (
                                    somMethod Tabs parentMtab,
                                     int parentNum,
                                     somMToken M Token);
```

Methods: somGetMethodData, **somGetNthMethodData**, **somGetRdStub**, **somAddStaticMethod**, **somAddDynamicMethod**

Description

The **somParentNumResolve** function is used to make parent method calls by the C and C++ language implementation bindings. The **somParentNumResolve** function returns a pointer to a procedure for performing the specified method. This pointer is selected from the specified method table, which is intended to be the method table corresponding to a parent class.

For C and C++ programmers, the implementation bindings for SOM classes provide convenient macros for making parent method calls (the "parent_" macros).

Parameters

parentMtab	A list of method tables for the parents of the class being implemented. The SOM API specifies that the list of parent method tables for a given class be stored in the auxiliary class data structure of the class, in the *parentMtab* component. Thus, for the class "XYZ", the parent method table list is found in location *XYZCClassData.parentMtab.* Parent method table lists are available from class objects by way of the method call **somGetPClsMtabs**.
parentNum	The position of the parent for which the method is to be resolved. The order of a class's parents is determined by the order in which they are specified in the interface statement for the class. (The first parent is number 1.)
mToken	The method token for the method to be resolved. The SOM API requires that if the class "XYZ" introduces the static method **foo**, then the method token for **foo** is found in the class data structure for "XYZ" (called XYZClassData) in the structure member named **foo** (that is, at XYZClassData.foo). Method tokens can also be obtained using the **somGetMethodToken** method.

Return Value

A **somMethodPtr** pointer to a procedure that implements the specified method, selected from the specified method table.

C++ Example

```
// SOM IDL for class A and class B
#include <somobj.idl>
module spnrExample {
   interface A : SOMObject  { void foo();  implementation {
                              callstyle=oidl; }; };
   interface B : A  {  implementation { foo: override; }; };
};

// Example C++ program to implement and test  module scrExample
#define SOM_Module_spnrexample_Source
#include <spnrExample.xih>
#include <stdio.h>

SOM_Scope void SOMLINK spnrExample_Afoo(spnrExample_A *somSelf);
{  printf("1\n"); }

SOM_Scope void SOMLINK spnrExample_Bfoo(spnrExample_B *somSelf);
{ printf("2\n"); }

main()
{
   spnrExample_B  *objPtr = new spnrExample_B;

   // This prints  2
   objPtr->foo();

   // This prints 1
   ((somTD_spnrExample_A_foo) /* This method procedure expression
cast
                                 is necessary */
      somParentNumResolve(
           objPtr->somGetClass()->somGetPClsMtabs(),
           1,
           spnrExample_AClassData.foo) // the foo method token
      ) /* end of method procedure expression */
      (objPtr); /* method arguments */
}
```

Related Information

Functions: somResolveByName, **somResolve**, **somParentNumResolve**, **somClassResolve**

Data Structures: somMethodPtr (**sombtype.h**), **somMethodTabs** (**somapi.h**), **somMToken** (**somapi.h**)

Methods: somGetPClsMtab, **somGetPClsMtabs**, **somGetMethodToken**

Macros: SOM_ParentNumResolve, **SOM_Resolve**, **SOM_ResolveNoCheck**

somParentResolve Function

Purpose

Obtains a pointer to a procedure that implements a method, given a list of method tables. Obsolete but still supported.

Syntax

```
somMethodPtr  somParentResolve (somMethodTabs parentMtab,
                                somMToken mToken);
```

Description

The **somParentResolve** function is used by old, single-parent class binaries to make parent method calls. The function is obsolete, but is still supported. The **somParentResolve** function returns a pointer to the procedure that implements the specified method. This pointer is selected from the first method table in the parentMtab list.

Parameters

parentMtab	A list of parent method tables, the first of which is the method table for the parent class for which the method is to be resolved. The SOM API specifies that the list of parent method tables for a given class be stored in the auxiliary class data structure of the class, in the *parentMtab* component. Thus, for the class "XYZ", the parent method table list is found in location XYZCClassData.parentMtab. Parent method table lists are available from class objects by way of the method call **somGetPClsMtabs**.
mToken	The method token for the method to be resolved. The SOM API requires that if the class "XYZ" introduces the static method "foo", then the method token for "foo" is found in the class data structure for "XYZ" (called XYZClassData) in the structure member named "foo" (that is, at XYZClassData.foo). Method tokens can also be obtained using the **somGetMethodToken** method.

Return Value

A **somMethodPtr** pointer to the procedure that implements the specified method, selected from the first method table.

Related Information

Functions: somResolveByName, **somResolve**, **somParentNumResolve**, **somClassResolve**

Data Structures: somMethodPtr (**sombtype.h**), **somMethodTabs** (**somapi.h**), **somMToken** (**somapi.h**)

Methods: somDispatch, somClassDispatch, **somFindMethod, somFindMethodOk**, **somGetApplyStub, somGetMethodToken**

Macros: SOM_Resolve, **SOM_ResolveNoCheck**

somPrefixLevel Function

Purpose

Outputs blanks to prefix a line at the indicated level.

Syntax

void somPrefixLevel (long *level***);**

Description

The **somPrefixLevel** function outputs blanks (through the **somPrintf** function) to prefix the next line of output at the indicated level. (The number of blanks produces is 2*level.) This function is useful when overriding the **somDumpSelfInt** method, which takes the level as an argument.

Parameters

level	The level at which the next line of output is to start.

C/C++ Example

```
#include <som.h>
somPrefixLevel(5);
```

Related Information

Functions: somPrintf, **somVprintf**, **somLPrintf**, **SOMOutCharRoutine**

somPrintf Function

Purpose

Prints a formatted string in the manner of the C printf function.

Syntax

long somPrintf (string *fmt,* **...);**

Description

The **somPrintf** function prints a formatted string using function **SOMOutCharRoutine**, in the same manner as the C printf function. The implementation of **SOMOutCharRoutine** determines the destination of the output, while the C printf function is always directed to stdout. (The default output destination for **SOMOutCharRoutine** is stdout also, but this can be modified by the user.)

Parameters

fmt	The format string to be output.
varargs	The values to be substituted into the format string.

Return Value

Returns the number of characters written.

C Example

```
#include <somcls.h>
somPrintf(”The class name is %s.\n”, _somGetClassName(obj));
```

Related Information

Functions: somVprintf, **somPrefixLevel**, **somLPrintf**, **SOMOutCharRoutine**

somRegisterId Function

Purpose

Registers a SOM ID and determines whether or not it was previously registered.

Syntax

int somRegisterId (somId *id)*;

Description

The **somRegisterId** function registers a SOM ID and converts it into an internal representation. If the ID is already registered, **somRegisterId** returns 0 and has no effect. Otherwise, **somRegisterId** returns 1.

Parameters

id	The **somId** to be registered.

Return Value

If the ID is already registered, **somRegisterId** returns 0. Otherwise, **somRegisterId** returns 1.

C Example

```
#include <som.h>
static string s = ”unregistered”;
static somId sid = &s;
main()
{
   somEnvironmentNew();
   SOM_Test(somRegisterId(sid) == 1);
   SOM_Test(somRegisterId(somIdFromString(”registered”)) == 0);
}
```

Related Information

Functions: somCheckId, **somIdFromString**, **somStringFromId**, **somCompareIds**, **somTotalRegIds**, **somSetExpectedIds**, **somUniqueKey**, **somBeginPersistentIds**, **somEndPersistentIds**

Data Structures: somId (**sombtype.h**)

somResolve Function

Purpose

Obtains a pointer to the procedure that implements a method for a particular SOM object.

Syntax

somMethodPtr somResolve (SOMObject obj**, somMToken** mToken**);**

Description

The **somResolve** function returns a pointer to the procedure that implements the specified method for the specified SOM object. This pointer can then be used to invoke the method. The **somResolve** function can only be used to obtain a method procedure for a static method (one declared in an IDL or OIDL specification for a class); dynamic methods are not supported by method tokens.

For C and C++ programmers, the SOM usage bindings for SOM classes provide more convenient mechanisms for invoking methods. These bindings use the **SOM_Resolve** and **SOM_ResolveNoCheck** macros, which construct a method token expression from the class name and method name, and call **somResolve**.

Parameters

obj	A pointer to the object whose method procedure is required.
mToken	The method token for the method to be resolved. The SOM API requires that if the class "XYZ" introduces the static method **foo**, then the method token for **foo** is found in the class data structure for "XYZ" (called XYZClassData) in the structure member named "foo" (that is, at XYZClassData.foo). Method tokens can also be obtained using the **somGetMethodToken** method.

Return Value

A **somMethodPtr** pointer to the procedure that implements the specified method for the specified SOM object.

C Example

```
// SOM IDL for class A and class B
#include <somobj.idl>
module srExample {
   interface A : SOMObject  { void foo();  implementation {
                              callstyle=oidl; }; };
   interface B : A  {  implementation { foo: override; }; };
};

// Example C++ program to implement and test  module scrExample
#define SOM_Module_srexample_Source
#include <srExample.ih>
#include <stdio.h>

SOM_Scope void SOMLINK srExample_Afoo(srExample_A *somSelf);
{  printf("1\n"); }

SOM_Scope void SOMLINK srExample_Bfoo(srExample_B *somSelf);
{ printf("2\n"); }

main()
{
   srExample_B  objPtr = srExample_BNew();

   /* This prints  2 */
   ((somTD_srExample_A_foo)   /* this method procedure expression
cast                                   is necessary */
       somResolve(objPtr, srExample_AClassData.foo)
       )     /* end of method procedure expression */
    (objPtr);
}
```

Related Information

Functions: somResolveByName, **somParentResolve**, **somParentNumResolve**, **somClassResolve**

Data Structures: somMethodPtr (**sombtype.h**), **somMToken** (**somapi.h**)

Methods: somDispatch, somClassDispatch, **somFindMethod, somFindMethodOk**, **somGetMethodToken**

Macros: SOM_Resolve, **SOM_ResolveNoCheck**

somResolveByName Function

Purpose

Obtains a pointer to the procedure that implements a method for a particular SOM object.

Syntax

somMethodPtr somResolveByName (SOMObject *obj,* **string** *methodName)*;

Description

The **somResolveByName** function is used to obtain a pointer to the procedure that implements the specified method for the specified SOM object. The returned procedure pointer can then be used to invoke the method. The C and C++ usage bindings use this function to support name-lookup methods.

This function can be used for invoking dynamic methods. However, the C and C++ usage bindings for SOM classes do not support dynamic methods, thus typedefs necessary for the use of dynamic methods are not available as with static methods. The function **somApply** provides an alternative mechanism for invoking dynamic methods that avoids the need for casting procedure pointers.

Parameters

obj	A pointer to the object whose method procedure is required.
methodName	A character string representing the name of the method to be resolved.

Return Value

A **somMethodPtr** pointer to the procedure that implements the specified method for the specified SOM object.

C Example

Assuming the static method “setSound,” is introduced by the class ”Animal”, the following example will correctly invoke this method on an instance of ”Animal” or one of its descendent classes.

```
#include <animal.h>
example(Animal myAnimal)
{
somTD_Animal_setSound
   setSoundProc = somResolveByName(myAnimal, ”setSound”);
setSoundProc(myAnimal, ”Roar!”);
}
```

Related Information

Functions: somResolve, **somParentResolve**, **somParentNumResolve**, **somClassResolve**

Data Structures: somMethodPtr (**sombtype.h**), **SOMObject** (**somobj.idl**), **string** (**somcorba.h**)

Methods: somDispatch, somClassDispatch, **somFindMethod, somFindMethodOk**, **somGetApplyStub**

Macros: SOM_Resolve, **SOM_ResolveNoCheck**

somSetException Function

Purpose

Sets an exception value in an **Environment** structure.

Syntax

void somSetException (Environment **ev,*
enum exception_type *major,*
string *exceptionName,*
somToken *params***);**

Description

The **somSetException** function sets an exception value in an **Environment** structure.

Parameters

ev	A pointer to the **Environment** structure in which to set the exception. This value must be either NULL or a value formerly obtained from the function **somGetGlobalEnvironment**.
major	An integer representing the type of exception to set.
exceptionName	The qualified name of the exception to set. The SOM Compiler defines, in the header files it generates for an interface, a constant whose value is the qualified name of each exception defined within the interface. This constant has the name "ex_<*exceptionName*>", where <*exceptionName*> is the qualified (scoped) exception name. Where unambiguous, the usage bindings also define the short form "ex_<exceptionName>", where <exceptionName> is unqualified.
params	A pointer to an initialized exception structure value. No copy is made of this structure; hence, the caller cannot free it. The **somExceptionFree** function should be used to free the **Environment** structure that contains it.

C Example

```
/* IDL declaration of class X:  */
   interface X : SOMObject {
      exception OUCH {long code1; long code2; };
      void foo(in long arg) raises (OUCH);
   };

/* implementation of foo method */
SOM_Scope void SOMLINK foo(X somSelf, Environment *ev, long arg)
{
   X_OUCH *exception_params; /* X_OUCH struct is defined
                                       in X's usage bindings
*/

   if (arg > 5) /* then this is a very bad error */
   {
      exception_params = (X_OUCH*)SOM_Malloc(sizeof(X_OUCH));
      exception_params->code1 = arg;
      exception_params->code2 = arg-5;
      somSetException(ev, USER_EXCEPTION, ex_X_OUCH,

                          exception_params);
      /* the Environment ev now contains an X_OUCH exception,
with
       * the specified exception_params struct. The constant
       * ex_X_OUCH is defined in foo.h. Note that
exception_params
       * must be malloced.
       */
      return;
    }
...
}

main()
{
   Environment *ev;
   X x;

   somEnvironmentNew();
   x = Xnew();
   ev = somGetGlobalEnvironment();
   X_foo(x, ev, 23);
   if (ev->_major != NO_EXCEPTION) {
        printf("foo exception = %s\n", somExceptionId(ev));
        printf("code1 = %d\n",
                ((X_OUCH*)somExceptionValue(ev))->code1);
        /* finished handling exception. */
        /* free the copied id and the original X_OUCH structure:
*/
        somExceptionFree(ev);
   }
...
}
```

Related Information

Functions: somExceptionId, **somExceptionValue**, **somExceptionFree**, **somGetGlobalEnvironment**

Data Structures: Environment, **exception_type**, **string** (**somcorba.h**)

somSetExpectedIds Function

Purpose

Tells SOM how many unique SOM IDs a client program expects to use.

Syntax

void somSetExpectedIds (**unsigned long** *numIds)*;

Description

The **somSetExpectedIds** function informs the SOM runtime environment how many unique SOM IDs a client program expects to use during its execution. This has the potential of slightly improving the program's space and time efficiency, if the value specified is accurate. This function, if used, must be called prior to any explicit or implicit invocation of the **somEnvironmentNew** function to have any effect.

Parameters

numIds	The number of SOM IDs the client program expects to use.

C Example

```
#include <som.h>
somSetExpectedIds(1000);
```

Related Information

Functions: somCheckId, **somRegisterId**, **somIdFromString**, **somStringFromId**, **somCompareIds**, **somTotalRegIds**, **somUniqueKey**, **somBeginPersistentIds**, **somEndPersistentIds**

somSetOutChar Function

Purpose

Changes the behavior of the **somPrintf** function.

Syntax

void somSetOutChar (
somTD_SOMOutCharRoutine * *outCharRtn***);**

Description

The **somSetOutChar** function is called to change the output character routine that **somPrintf** invokes. By default, **somPrintf** invokes a character output routine that goes to **stdout**.

The execution of **somSetOutChar** affects only the application (or thread) in which it occurs. Thus, **somSetOutChar** is normally preferred over **SOMOutCharRoutine** for changing the output routine called by **somPrintf**, since **SOMOutCharRoutine** remains in effect for subsequent threads as well.

Some additional samples of **somSetOutChar** can be found in the **somapi.h** header file.

Parameters

outCharRtn	A pointer to your routine that outputs a character in the way you want.

Example

```
#include <som.h>
static int irOutChar(char c);

static int irOutChar(char c)
{
    (Customized code goes here)
}

main (...)
{
    ...
    somSetOutChar((somTD_SOMOutCharRoutine *) irOutChar);
}
```

Related Information

Functions: somPrintf, SOMOutCharRoutine

somStringFromId Function

Purpose

Returns the string that a SOM ID represents.

Syntax

string somStringFromId (somId *id***)**;

Description

The **somStringFromId** function returns the string that a given SOM ID represents.

Parameters

id	The SOM ID for which the corresponding string is needed.

Return Value

Returns the string that the given SOM ID represents.

Example

See the **somBeginPersistentIds** function.

Related Information

Functions: somCheckId, **somRegisterId**, **somIdFromString**, **somCompareIds**, **somTotalRegIds**, **somSetExpectedIds**, **somUniqueKey**, **somBeginPersistentIds**, **somEndPersistentIds**

Data Structures: string (**somcorba.h**), **somId** (**sombtype.h**)

somTotalRegIds Function

Purpose

Returns the total number of SOM IDs that have been registered.

Syntax

unsigned long somTotalRegIds ( );

Description

The **somTotalRegIds** function returns the total number of SOM IDs that have been registered so far. This value can be used as a parameter to the **somSetExpectedIds** function to advise SOM about expected ID usage in later executions of a client program.

Return Value

Returns the total number of SOM IDs that have been registered.

C Example

```
#include <som.h>
main()
{ int i;
  somId id;
  somEnvironmentNew();
  id = somIdFromString(”abc”)
  i = somTotalRegIds();
  id = somIdFromString(”abc”);
  SOM_Test(i == somTotalRegIds);
}
```

Related Information

Functions: somCheckId, **somRegisterId**, **somIdFromString**, **somStringFromId**, **somCompareIds**, **somSetExpectedIds**, **somUniqueKey**, **somBeginPersistentIds**, **somEndPersistentIds**

somUniqueKey Function

Purpose

Returns the unique key associated with a SOM ID.

Syntax

unsigned long somUniqueKey (somID *id)*;

Description

The **somUniqueKey** function returns the unique key associated with a SOM ID. The unique key for a SOM ID is a number that uniquely represents the string that the SOM ID represents. The unique key for a SOM ID is the same as the unique key for another SOM ID only if the two SOM IDs represent the same string.

Parameters

id	The SOM ID for which the unique key is needed.

Return Value

An **unsigned long** representing the unique key of the specified SOM ID.

C Example

```
#include <som.h>
main()
{
   unsigned long k1, k2;
   k1 = somUniqueKey(somIdFromString(”abc”));
   k2 = somUniqueKey(somIdFromString(”abc”));
   SOM_Test(k1 == k2);
}
```

Related Information

Functions: somCheckId, **somRegisterId**, **somIdFromString**, **somStringFromId**, **somCompareIds**, **somTotalRegIds**, **somSetExpectedIds**, **somBeginPersistentIds**, **somEndPersistentIds**

Data Structures: somId (**sombtype.h**)

somVprintf Function

Purpose

Prints a formatted string in the manner of the C vprintf function.

Syntax

long somVprintf (string *fmt*, **va_list** *ap***)**;

Description

The **somVprintf** function prints a formatted string using **SOMOutCharRoutine**, in the same manner as the C vprintf function. The implementation of **SOMOutCharRoutine** determines the destination of the output, while the C printf function is always directed to stdout. (The default output destination for **SOMOutCharRoutine** is stdout also, but this can be modified by the user.)

Parameters

fmt	The format string to be output.
ap	A **va_list** representing the values to be substituted into the format string.

Return Value

Returns the number of characters written.

C Example

```
#include <som.h>
main()
{
   va_list args = (va_list) SOMCalloc(20);
   va_list push = args;
   float f = 3.1415
   char c = ’a’;

   va_arg(push, int) = 1;
   va_arg(push, double) = f; /* note ANSI widening */
   va_arg(push, int) = c; /* here, too */
   va_arg(push, char*) = ”this is a test”;

   somVprintf(”%d, %f, %c, %s\n”, args);
}
```

Related Information

Functions: somPrintf, **somPrefixLevel**, **somLPrintf**, **SOMOutCharRoutine**

Data Structures: string (**somcorba.h**), **va_list** (**stdarg.h**)

SOMCalloc Function

Purpose

Allocates sufficient zeroed memory for an array of objects of a specified size.

Syntax

somToken (*SOMCalloc) (size_t *num*, **size_t** *size***);**

Description

The **SOMCalloc** function allocates an amount of memory equal to *num*size* (sufficient memory for an array of *num* objects of size *size).* The **SOMCalloc** function has the same interface as the C **calloc** function. It performs the same basic function as **calloc** with some supplemental error checking. If an error occurs, the **SOMError** function is called. This routine is replaceable by changing the value of the global variable **SOMCalloc.**

Parameters

num	The number of objects for which space is to be allocated.
size	The size of the objects for which space to is to be allocated.

Return Value

A pointer to the first byte of the allocated space.

Example

See the **somVprintf** function.

Related Information

Functions: SOMMalloc, **SOMRealloc**, **SOMFree**

Data Structures: somToken (**somapi.h**)

SOMClassInitFuncName Function

Purpose

Returns the name of the function used to initialize classes in a DLL.

Syntax

string (*SOMClassInitFuncName) ( );

Description

The **SOMClassInitFuncName** function is called by the SOM Class Manager to determine what function to call to initialize the classes in a DLL. The default version returns the string "SOMInitModule." The function can be replaced (so that the Class Manager will invoke a different function to initialize classes in a DLL) by changing the value of the global variable **SOMClassInitFuncName**.

Return Value

Returns the name of the function that should be used to initialize classes in a DLL.

C Example

```
#include <som.h>
string XYZFuncName() { return "XYZ"; }
main()
{
   SOMClassInitFuncName = XYZFuncName;
   ...
}
```

Related Information

Functions: SOMLoadModule, **SOMDeleteModule**

Data Structures: string (**somcorba.h**)

SOMDeleteModule Function

Purpose

Unloads a dynamically linked library (DLL).

Syntax

int (*SOMDeleteModule) (somToken *modHandle***);**

Description

The **SOMDeleteModule** function unloads the specified dynamically linked library (DLL). This routine is called by the SOM Class Manager to unload DLLs. **SOMDeleteModule** can be replaced (thus changing the way the Class Manager unloads DLLS) by changing the value of the global variable **SOMDeleteModule.**

Parameters

modHandle	The **somToken** for the DLL to be unloaded. This token is supplied by the **SOMLoadModule** function when it loads the DLL.

Return Value

Returns 0 if successful or a non-zero system-specific error code otherwise.

Related Information

Functions: SOMLoadModule, **SOMClassInitFuncName**

Data Structures: somToken (**somapi.h**)

SOMError Function

Purpose

Handles an error condition.

Syntax

void (*SOMError) (int *errorCode*, **string** *fileName*, **int** *lineNum***);**

Description

The **SOMError** function inspects the specified error code and takes appropriate action, depending on the severity of the error. The last digit of the error code indicates whether the error is classified as SOM_Fatal (9), SOM_Warn (2), or SOM_Ignore (1). The default implementation of **SOMError** prints a message that includes the specified error code, filename, and line number, and terminates the current process if the error is classified as SOM_Fatal. The *fileName* and *lineNum* arguments specify where the error occurred. This routine can be replaced by changing the value of the global variable **SOMError.**

For C and C++ programmers, SOM defines a convenience macro, **SOM_Error**, which invokes the **SOMError** function and supplies the last two arguments.

Parameters

errorCode	An integer representing the error code of the error.
fileName	The name of the file in which the error occurred.
lineNum	The line number where the error occurred.

Related Information

Macros: SOM_Test, **SOM_TestC**, **SOM_WarnMsg**, **SOM_Assert**, **SOM_Expect**, **SOM_Error**

SOMFree Function

Purpose

Frees the specified block of memory.

Syntax

void (*SOMFree) (somToken *ptr***);**

Description

The **SOMFree** function frees the block of memory pointed to by *ptr.* SOMFree should only be called with a pointer previously allocated by **SOMMalloc** or **SOMCalloc.** The **SOMFree** function has the same interface as the C **free** function. It performs the same basic function as **free** with some supplemental error checking. If an error occurs, the **SOMError** function is called. This routine is replaceable by changing the value of the global variable **SOMFree.**

To free an *object* (rather than a block of memory), use the **somFree** method, rather than this function.

Parameters

ptr	A pointer to the block of storage to be freed.

C Example

```
#include <som.h>
main()
{
   somToken ptr = SOMMalloc(20);
   . . .
   somFree(ptr);
}
```

Related Information

Functions: SOMCalloc, **SOMMalloc**, **SOMRealloc**

Methods: somFree

SOMInitModule Function

Purpose

Invokes the class creation routines for the classes contained in an OS/2 or Windows class library (DLL).

Syntax

SOMEXTERN void SOMLINK SOMInitModule (
long *MajorVersion*,
long *MinorVersion*,
string *ClassName***);**

Description

On OS/2 or Windows, a class library (DLL) can contain the implementations for multiple classes, all of which should be created when the DLL is loaded. On OS/2, when loading a DLL, the SOM class manager determines the name of a DLL initialization function, and if the DLL exports a function of this name, the class manager invokes that function (whose purpose is to create the classes in the DLL). **SOMInitModule** is the default name for this DLL initialization function.

On Windows, the SOM class manager does *not* call **SOMInitModule**. It must be called from the default Windows DLL initialization function, **LibMain.** This call is made indirectly through the **SOM_ClassLibrary** macro (see the example that follows).

Parameters

MajorVersion	The major version number of the class that was requested when the library was loaded.
MinorVersion	The minor version number of the class that was requested when the library was loaded.
ClassName	The name of the class that was requested when the library was loaded.

Example

```
#include ”xyz.h”
#ifdef __IBMC__
  #pragma linkage (SOMInitModule, system)
#endif

SOMEXTERN void  SOMLINK SOMInitModule (long majorVersion,
                          long minorVersion, string className)
{
    SOM_IgnoreWarning (majorVersion);  /* This function makes  */
    SOM_IgnoreWarning (minorVersion);  /* no use of the passed */
    SOM_IgnoreWarning (className);     /* arguments.   */
    xyzNewClass (A_MajorVersion, A_MinorVersion);
}
```

For Windows, also include the following function:

```
#include <windows.h>
int CALLBACK LibMain (HINSTANCE inst,
                          WORD ds,
                          WORD Heapsize,
                          LPSTR cmdLine)
{
        SOM_IgnoreWarning (inst);
        SOM_ignoreWarning (ds);
        SOM_IgnoreWarning (heapSize);
        SOM_IgnoreWarning (cmdLine);

        SOM_ClassLibrary (”xyz.dll”);
        return 1;  /* Indicate success to loader */
}
```

Related Information

Functions: SOMClassInitFuncName

Methods: somGetInitFunction

Macros: SOM_ClassLibrary

SOMLoadModule Function

Purpose

Loads the dynamically linked library (DLL) containing a SOM class.

Syntax

int (*SOMLoadModule) (
 string *className*,
 string *fileName,*
 string *functionName,*
 long *majorVersion,*
 long *minorVersion,*
 somToken ***modHandle**);

Description

The **SOMLoadModule** function loads the dynamically linked library (DLL) containing a SOM class. This routine is called by the SOM Class Manager to load DLLs. **SOMLoadModule** can be replaced (thus changing the way the Class Manager loads DLLS) by changing the value of the global variable **SOMLoadModule.**

Parameters

className	The name of the class whose DLL is to be loaded.
fileName	The name of the DLL library file. This can be either a simple name or a fully qualified pathname.
functionName	The name of the routine to be called after the DLL is loaded. The routine is responsible for creating a class object for each class in the DLL. Typically, this argument will have the value **SOMInitModule,** obtained from the **SOMClassInitFuncName** function. If no **SOMInitModule** entry exists in the DLL, the default version of **SOMLoadModule** looks for a routine named <*className*>**NewClass** instead. If neither entry point is found, the default version of **SOMLoadModule** fails.
majorVersion	The expected major version number of the class, to be passed to the initialization routine of the DLL.
minorVersion	The expected minor version number of the class, to be passed to the initialization routine of the DLL.
modHandle	The address where **SOMLoadModule** should place a token that can be subsequently used by the **SOMDeleteModule** routine to unload the DLL.

Return Value

Returns 0 if successful or a non-zero system-specific error code otherwise.

Related Information

Functions: SOMDeleteModule, **SOMClassInitFuncName**

SOMMalloc Function

Purpose

Allocates the specified amount of memory.

Syntax

somToken (*SOMMalloc) (size_t *size***);**

Description

The **SOMMalloc** function allocates *size* bytes of memory. The **SOMMalloc** function has the same interface as the C **malloc** function. It performs the same basic function as **malloc** with some supplemental error checking. If an error occurs, the **SOMError** function is called. This routine is replaceable by changing the value of the global variable **SOMMalloc.**

Parameters

size	The amount of memory to be allocated, in bytes.

Return Value

A pointer to the first byte of the allocated space.

Example

See the **SOMFree** function.

Related Information

Functions: SOMCalloc, **SOMRealloc**, **SOMFree**

SOMOutCharRoutine Function

Purpose

Prints a character. This function is replaceable.

Syntax

int (*SOMOutCharRoutine) (char *c***);**

Description

SOMOutCharRoutine is a replaceable character output routine. It is invoked by SOM whenever a character is generated by one of the SOM error-handling or debugging macros. The default implementation outputs the specified character to stdout. To change the destination of character output, store the address of a user-written character output routine in global variable **SOMOutCharRoutine**.

Another function, **somSetOutChar**, may be preferred over the **SOMOutCharRoutine** function. The **somSetOutChar** function enables each application (or thread) to have a customized character output routine.

Parameters

c	The character to be output.

Return Value

Returns 0 if an error occurs and 1 otherwise.

Example

```
#include <som.h>
#pragma linkage(myCharacterOutputRoutine, system)
/* Define a replacement routine: */
int SOMLINK myCharacterOutputRoutine (char c)
{
    (Customized code goes here)
}
...
/* After the next stmt all output */
/* will be sent to the new routine   */
SOMOutCharRoutine = myCharacterOutputRoutine;
```

Related Information

Functions: somVprintf, **somPrefixLevel**, **somLPrintf**, **somPrintf**, **somSetOutChar**

SOMRealloc Function

Purpose

Changes the size of a previously allocated region of memory.

Syntax

somToken (*SOMRealloc) (somToken *ptr,* **size_t** *size***);**

Description

The **SOMRealloc** function changes the size of the previously allocated region of memory pointed to by *ptr* so that it contains *size* bytes. The new size may be greater or less than the original size. The **SOMRealloc** function has the same interface as the C **realloc** function. It performs the same basic function as **realloc** with some supplemental error checking. If an error occurs, the **SOMError** function is called. This routine is replaceable by changing the value of the global variable **SOMRealloc.**

Parameters

ptr	A pointer to the previously allocated region of memory. If NULL, a new region of memory of *size* bytes is allocated.
size	The size in bytes for the re-allocated storage. If zero, the memory pointed to by *ptr* is freed.

Return Value

A pointer to the first byte of the re-allocated space. (A pointer is returned because the block of storage may need to be moved to increase its size).

Related Information

Functions: SOMCalloc, **SOMMalloc**, **SOMFree**

SOM_Assert Macro

Purpose

Asserts that a **boolean** condition is true.

Syntax

void SOM_Assert (
    boolean *condition,*
    long *errorCode***);**

Description

The **SOM_Assert** macro is used to place **boolean** assertions in a program:

- If *condition* is FALSE, and *errorCode* indicates a warning-level error and **SOM_WarnLevel** is set to be greater than zero, then a warning message is output.
- If *condition* is FALSE and *errorCode* indicates a fatal error, an error message is output and the process is terminated.
- If *condition* is TRUE and **SOM_AssertLevel** is set to be greater than zero, then an informational message is output.

External (Global) Data

```
long SOM_WarnLevel;  /* default = 0 */

long SOM_AssertLevel; /* default 0 */
```

Parameters

condition	A **boolean** expression that is expected to be TRUE (nonzero).
errorCode	The integer error code for the error to be raised if *condition* is FALSE.

Expansion

If *condition* is FALSE, and *errorCode* indicates a warning-level error and **SOM_WarnLevel** is set to be greater than zero, then a warning message is output. If *condition* is FALSE and *errorCode* indicates a fatal error, an error message is output and the process is terminated. If *condition* is TRUE and **SOM_AssertLevel** is set to be greater than zero, then an information message is output.

Example

```
#include <som.h>
main()
{
   SOM_WarnLevel = 1;
   SOM_Assert(2==2, 29);
}
```

Related Information

Macros: SOM_Expect, **SOM_Test**, **SOM_TestC**

SOM_ClassLibrary Macro

Purpose

Identifies the file name of the DLL for a SOM class library in a Windows LibMain function.

Syntax

void SOM_ClassLibrary (string "*libname.dll* **");**

Description

Each Windows SOM class library must supply a Windows LibMain function. In LibMain, the **SOM_ClassLibrary** macro identifies both the actual file name of the library as it would appear in a Windows LoadLibrary call and the location of the library's **SOMInitModule** function. This information is passed to the SOM Kernel, which in turn registers the library and schedules the execution of the **SOMInitModule** function. This macro can also be used in OS/2 class libraries within the context of a DLL "init/term" function.

Typically, the SOM Kernel invokes the **SOMInitModule** function of each statically loaded class library during the execution of the **somMainProgram** function in the using application. For dynamically loaded class libraries, **SOMInitModule** is invoked immediately upon completion of the library's LibMain (or an OS/2 DLL "init/term") function.

Because the **SOM_ClassLibrary** macro expands to reference the **SOMInitModule** function, either a declaration of the **SOMInitModule** function, or the function itself, should precede the appearance of **SOM_ClassLibrary** in the current compilation unit, as shown in the following example).

Parameters

libname.dll	The name of the file containing the DLL (as the name would appear in a Windows LoadLibrary call).

Example

```
/* This example illustrates the use of the SOM_ClassLibrary
   macro in a Windows LibMain function */

#include <som.h>
SOMEXTERN void SOMLINK SOMInitModule (long majorVersion,
                                      long minorVersion,
                                      string className);

#include <windows.h>
int CALLBACK LibMain (HINSTANCE inst,
                      WORD ds,
                      WORD Heapsize,
                      LPSTR cmdLine)
{
     SOM_IgnoreWarning (inst);
     SOM_ignoreWarning (ds);
     SOM_IgnoreWarning (heapSize);
     SOM_IgnoreWarning (cmdLine);

     SOM_ClassLibrary ("xyz.dll");
     return 1;  /* Indicate success to loader */
}
```

Related Information

Macros: SOM_MainProgram

Functions: somMainProgram

SOM_CreateLocalEnvironment Macro

Purpose

Creates and initializes a local **Environment** structure.

Syntax

Environment * SOM_CreateLocalEnvironment ( );

Description

The **SOM_CreateLocalEnvironment** macro creates a local **Environment** structure. This **Environment** structure can be passed to methods as the **Environment** argument so that exception information can be returned without affecting the global environment.

Expansion

The **SOM_CreateLocalEnvironment** expands to an expression of type (**Environment** *).

C Example

```
Environment *ev;
ev = SOM_CreateLocalEnvironment();
_myMethod(obj, ev);
...
SOM_DestroyLocalEnvironment(ev);
```

Related Information

Macros: SOM_DestroyLocalEnvironment, **SOM_InitEnvironment**, **SOM_UninitEnvironment**

Data Structures: Environment (**somcorba.h**)

Functions: somGetGlobalEnvironment

SOM_DestroyLocalEnvironment Macro

Purpose

Destroys a local **Environment** structure.

Syntax

void SOM_DestroyLocalEnvironment (Environment * *ev***);**

Description

The **SOM_DestroyLocalEnvironment** macro destroys a local **Environment** structure, such as one created using the **SOM_CreateLocalEnvironment** macro.

Parameters

ev	A pointer to the **Environment** structure to be discarded.

Expansion

The **SOM_DestroyLocalEnvironment** function first invokes the **somExceptionFree** function on the **Environment** structure; then it invokes **SOMFree** on it to free the memory it occupies.

Example

```
Environment *ev;
ev = SOM_CreateLocalEnvironment();
_myMethod(obj, ev);
....
SOM_DestroyLocalEnvironment(ev);
```

Related Information

Macros: SOM_CreateLocalEnvironment, **SOM_UninitEnvironment**

Functions: somExceptionFree

SOM_Error Macro

Purpose

Reports an error condition.

Syntax

void SOM_Error (long *errorCode***);**

Description

The **SOM_Error** macro invokes the **SOMError** error handling procedure with the specified error code, supplying the filename and line number where the macro was invoked. The default implementation of **SOMError** outputs a message containing the error code, filename, and line number. Additionally, if the last digit of the error code indicates a serious error (that is, value SOM_Fatal), the process is terminated.

Parameters

errorCode	The integer error code for the error to be reported.

Expansion

The **SOM_Error** macro invokes the **SOMError** error handler, supplying the filename and line number where the macro was invoked.

Related Information

Functions: SOMError

SOM_Expect Macro

Purpose

Asserts that a **boolean** condition is expected to be true.

Syntax

void SOM_Expect (boolean *condition***);**

Description

The **SOM_Expect** macro is used to place **boolean** assertions that are expected to be true into a program:

- If *condition* is FALSE and **SOM_WarnLevel** is set to be greater than zero, then a warning message is output.
- If *condition* is TRUE and **SOM_AssertLevel** is set to be greater than zero, then an informational message is output.

Parameters

condition	A boolean expression that is expected to be TRUE (nonzero).

Expansion

If *condition* is FALSE and **SOM_WarnLevel** is set to be greater than zero, then a warning message is output. If *condition* is TRUE and **SOM_AssertLevel** is set to be greater than zero, then an information message is output.

Example

```
SOM_Expect(2==2);
```

Related Information

Macros: SOM_Assert, **SOM_Test**, **SOM_TestC**

SOM_GetClass Macro

Purpose

Returns a pointer to the class object of which a SOM object is an instance.

Syntax

SOMClass SOM_GetClass (SOMObject *objPtr***);**

Description

The **SOM_GetClass** macro returns the class object of which *obj* is an instance. This is done without recourse to a method call on the object. The **somGetClass** method introduced by **SOMObject** is also intended to return the class of which an object is an instance, and the default implementation provided for this method by **SOMObject** uses the macro.

Important Note: It is generally recommended that the **somGetClass** method call be used, since it cannot be known whether the class of an object wishes to provide special handling when its address is requested from an instance. But, there are (rare) situations where a method call cannot be made, and this macro can then be used. If you are unsure as to whether to use the method or the macro, you should use the method.

Parameters

objPtr	A pointer to the object whose class is needed.

C++ Example

```
#include <somcls.xh>
#include <animal.xh>
main()
{
   Animal *a = new Animal;
   SOMClass cls1 = SOM_GetClass(a);
   SOMClass cls2 = a->somGetClass();
   if (cls1 == cls2)
      printf(”macro and method for getClass the same for
Animal\n”);
   else
      printf(”macro and method for getClass not same for
Animal\n”);
}
```

Related Information

Methods: somGetClass

SOM_InitEnvironment Macro

Purpose

Initializes a local **Environment** structure.

Syntax

void SOM_InitEnvironment (Environment * *ev***);**

Description

The **SOM_InitEnvironment** macro initializes a locally declared **Environment** structure. This **Environment** structure can then be passed to methods as the **Environment** argument so that exception information can be returned without affecting the global environment.

Parameters

ev	A pointer to the **Environment** structure to be initialized.

Expansion

The **SOM_InitEnvironment** initializes an **Environment** structure to zero.

C Example

```
Environment ev;
SOM_InitEnvironment(&ev);
_myMethod(obj, &ev);
....
SOM_UninitEnvironment(&ev);
```

Related Information

Macros: SOM_DestroyLocalEnvironment, **SOM_CreateLocalEnvironment**, **SOM_UninitEnvironment**

Functions: somGetGlobalEnvironment

SOM_MainProgram Macro

Purpose

Identifies an application as a SOM program and registers an end-of-program exit procedure to release SOM resources when the application terminates.

Syntax

SOMClassMgr SOM_MainProgram ( );

Description

The **SOM_MainProgram** macro should appear near the beginning of each Windows application program that uses SOM or a SOM class library. It can also be used in OS/2 or AIX programs but is not generally required on these platforms. Any statically referenced SOM class libraries are initialized during the execution of this macro, and an end-of-program exit procedure is established to release SOM resources during normal program termination. (This macro combines the execution of the C/C++ "atexit" function with the SOM **somMainProgram** function and returns a reference to the global **SOMClassMgr** object.)

Example

```
#include <som.h>
#include <windows.h>

int PASCAL WinMain (HINSTANCE inst,
                    WORD ds,
                    WORD Heapsize,
                    LPSTR cmdLine)
{
     ...
     SOM_MainProgram ();
     ...
     /* Rest of main program follows */
}
```

Related Information

Functions: somMainProgram

Macros: SOM_ClassLibrary

SOM_NoTrace Macro

Purpose

Used to turn off method debugging.

Syntax

SOM_NoTrace (<token> *className*, **<token>** *methodName***);**

Description

The **SOM_NoTrace** macro is used to turn off method debugging. Within an implementation file for a class, before #including the implementation (.ih or .xih) header file for the class, #define the <*className*>**MethodDebug** macro to be **SOM_NoTrace**. Then, <*className*>**MethodDebug** will have no effect.

Parameters

className	The name of the class for which tracing will be turned off, given as a simple token rather than a quoted string.
methodName	The name of the method for which tracing will be turned off, given as a simple token rather than a quoted string.

Expansion

The **SOM_NoTrace** macro has a null (empty) expansion.

Example

Within an implementation file:

```
#define AnimalMethodDebug(c,m) SOM_NoTrace(c,m)
#include <animal.ih>
/* Now AnimalMethodDebug does nothing */
```

SOM_ParentNumResolve Macro

Purpose

Obtains a pointer to a method procedure from a list of method tables. Used by C and C++ implementation bindings to implement parent method calls.

Syntax

somMethodPtr SOM_ParentNumResolve (
<token> *IntroClass,*
long *parentNum,*
somMethodTabs *parentMtabs,*
<token> *methodName***);**

Description

The **SOM_ParentNumResolve** macro invokes the **somParentNumResolve** function to obtain a pointer to the static method procedure that implements the specified method for the specified parent. The method is specified by indicating the introducing class, *IntroClass*, and the method name, *methodName*.

Parameters

introClass	The name of the class that introduces *methodName*. This name should be given as a simple token, rather than a quoted string (for example, *Animal* rather than "*Animal*").
parentNum	The position of the desired parent. The first (leftmost) parent of a class has position 1.
parentMtabs	A list of parent method tables acquired by invoking the **somGetPClsMtabs** method on a class object.
methodName	The name of the method to be resolved. This name should be given as a simple token, rather than a quoted string (for example, *setSound* rather than "*setSound*").

Expansion

The expansion of the macro produces an expression that is appropriately typed for application of the evaluated result to the indicated method's arguments, as illustrated in the following example.

Example

```
#include <somcls.h>

main()
{
 SOMClassMgr *cm = somEnvironmentNew();
 somMethodTabs mList = _somGetPClsMtabs(_SOMClass);
 SOM_ParentNumResolve(SOMObject, 1, mList, somDumpSelfInt)
      (_SOMClass,1);
}
```

Related Information

Functions: somParentNumResolve

Methods: somGetPClsMtabs

SOM_Resolve Macro

Purpose

Obtains a pointer to a static method procedure.

Syntax

somMethodPtr SOM_Resolve (
SOMObject *objPtr,*
<token> *className,*
<token> *methodName***);**

Description

The **SOM_Resolve** macro invokes the **somResolve** function to obtain a pointer to the static method procedure that implements the specified method for the specified object. This pointer can be used for efficient repeated casted method invocations on instances of the class of the object on which the resolution is done, or instances of subclasses of this class. The name of the class that introduces the method and the name of the method must be known to use this macro. Otherwise, use the **somResolveByName, somFindMethod** or **somFindMethodOk** method.

The **SOM_Resolve** macro can only be used to obtain a method procedure for a static method (one defined in the IDL specification for a class); not a dynamic method. Unlike the **SOM_ResolveNoCheck** macro, the **SOM_Resolve** macro performs several consistency checks on the object pointed to by *objPtr.*

Parameters

objPtr	A pointer to the object to which the resolved method procedure will be applied.
className	The name of the class that introduces *methodName*. This name should be given as a simple token, rather than a quoted string (for example, *Animal* rather than "*Animal*").
methodName	The name of the method to be resolved. This name should be given as a simple token, rather than a quoted string (for example, *setSound* rather than "*setSound*").

Expansion

The **SOM_Resolve** macro uses the *className* and *methodName* to construct the method token for the specified method, then invokes the **somResolve** function. Thus, the macro expands to an expression that represents the entry-point address of the method procedure. This value can be stored in a variable and used for subsequent invocations of the method.

Example

```
Animal myObj = AnimalNew();
somMethodProc *procPtr;
procPtr = SOM_Resolve(myObj, Animal, setSound);
/* note that procPtr will need to be typecast when it is used */
```

Related Information

Macros: SOM_ResolveNoCheck

Functions: somResolve, **somClassResolve**, **somResolveByName**

Methods: somFindMethod, somFindMethodOk, **somDispatch, somClassDispatch**

SOM_ResolveNoCheck Macro

Purpose

Obtains a pointer to a static method procedure, without doing consistency checks.

Syntax

somMethodPtr SOM_ResolveNoCheck (
SOMObject *objPtr,*
<token> *className,*
<token> *methodName***);**

Description

The **SOM_ResolveNoCheck** macro invokes the **somResolve** function to obtain a pointer to the method procedure that implements the specified method for the specified object. This pointer can be used for efficient repeated invocations of the same method on the same type of objects. The name of the class that introduces the method and the name of the method must be known at compile time. Otherwise, use the **somFindMethod** or **somFindMethodOk** method.

The **SOM_ResolveNoCheck** macro can only be used to obtain a method procedure for a static method (one defined in the IDL specification for a class) and not a method added to a class at run time. Unlike the **SOM_Resolve** macro, the **SOM_ResolveNoCheck** macro does not perform any consistency checks on the object pointed to by *objPtr.*

Parameters

objPtr	A pointer to the object to which the resolved method procedure will be applied.
className	The name of the class that introduces *methodName*. This name should be given as a simple token, rather than a quoted string (for example, *Animal* rather than "*Animal*").
methodName	The name of the method to be resolved. This name should be given as a simple token, rather than a quoted string (for example, *setSound* rather than "*setSound*").

Expansion

The **SOM_ResolveNoCheck** macro uses the *className* and *methodName* to construct an expression whose value is the method token for the specified method, then invokes the **somResolve** function. Thus, the macro expands to an expression that represents the entry-point address of the method procedure. This value can be stored in a variable and used for subsequent invocations of the method.

Example

```
Animal myObj = AnimalNew();
somMethodProc *procPtr;
procPtr = SOM_ResolveNoCheck(myObj, Animal, setSound)
```

Related Information

Macros: SOM_Resolve

Functions: somResolve, **somClassResolve**, **somResolveByName**

Methods: somDispatch, somClassDispatch, **somFindMethod, somFindMethodOk**

SOM_SubstituteClass Macro

Purpose

Provides a convenience macro for invoking the **somSubstituteClass** method.

Syntax

long SOM_SubstituteClass (
<token> *oldClass*,
<token> *newClass*);

Description

The method **somSubstituteClass** requires existing class objects as arguments. Therefore, the macro **SOM_SubstituteClass** first assures that the classes named *oldClass* and *newClass* exist, and then calls the method **somSubstituteClass** with these class objects as arguments.

Parameters

oldClass	The name of the class to be substituted, given as a simple token rather than a quoted string.
newClass	The name of the class that will replace *oldClass*, given as a simple token rather than a quoted string.

Example

See the method **somSubstituteClass**.

Related Information

Methods: somSubstituteClass

SOM_Test Macro

Purpose

Tests whether a **boolean** condition is true; if not, a fatal error is raised.

Syntax

void SOM_Test (boolean *expression***);**

Description

The **SOM_Test** macro tests the specified **boolean** expression:

- If the expression is TRUE and **SOM_AssertLevel** is set to a value greater than zero, then an information message is output.
- If the expression is FALSE, an error message is output and the process is terminated.

  Note: The **SOM_TestC** macro is similar, except that it only outputs a warning message in this situation.

Parameters

expression	The **boolean** expression to test.

External (Global) Data

```
long SOM_AssertLevel;  /* default is 0 */
```

Expansion

The **SOM_Test** macro tests the specified boolean expression. If the expression is TRUE and **SOM_AssertLevel** is set to a value greater than zero, then an information message is output. If the expression is FALSE, an error message is output and the process is terminated.

C Example

```
#include <som.h>
main()
{
   SOM_AssertLevel = 1;
   SOM_Test(1=1);
}
```

Related Information

Macros: SOM_Assert, **SOM_Except**, **SOM_TestC**

SOM_TestC Macro

Purpose

Tests whether a **boolean** condition is true; if not, a warning message is output.

Syntax

void SOM_TestC (boolean *expression***);**

Description

The **SOM_TestC** macro tests the specified **boolean** expression:

- If the expression is TRUE and **SOM_AssertLevel** is set to a value greater than zero, then an information message is output.
- If the expression is FALSE and **SOM_WarnLevel** is set to a value greater than zero, then a warning message is output.

  Note: The **SOM_Test** macro is similar, except that it raises a fatal error in this situation.

Parameters

expression	The **boolean** expression to test.

External (Global) Data

```
long SOM_AssertLevel;  /* default is 0 */

long SOM_WarnLevel; /* default is 0 */
```

Expansion

The **SOM_TestC** macro tests the specified **boolean** expression. If the expression is TRUE and **SOM_AssertLevel** is set to a value greater than zero, then an information message is output. If the expression is FALSE and **SOM_WarnLevel** is set to a value greater than zero, a warning message is output.

C Example

```
#include <som.h>
main()
{
   SOM_WarnLevel = 1;
   SOM_TestC(1=1);
}
```

Related Information

Macros: SOM_Assert, **SOM_Except**, **SOM_Test**

SOM_UninitEnvironment Macro

Purpose

Uninitializes a local **Environment** structure.

Syntax

void SOM_UninitEnvironment (Environment * *ev***);**

Description

The **SOM_UninitEnvironment** macro uninitializes a locally declared **Environment** structure.

Parameters

ev	A pointer to the **Environment** structure to be uninitialized.

Expansion

The **SOM_UninitEnvironment** invokes the **somExceptionFree** function on the specified **Environment** structure.

C Example

```
Environment ev;
SOM_InitEnvironment(&ev);
_myMethod(obj, &ev);
...
SOM_UninitEnvironment(&ev);
```

Related Information

Macros: SOM_DestroyLocalEnvironment, **SOM_InitEnvironment**

SOM_WarnMsg Macro

Purpose

Reports a warning message.

Syntax

void SOM_WarnMsg (string *msg***);**

Description

If **SOM_WarnLevel** is set to a value greater than zero, the **SOM_WarnMsg** macro prints the specified message, along with the filename and line number where the macro was invoked.

Parameters

msg	The warning message to be output.

Expansion

If **SOM_WarnLevel** is set to a value greater than zero, the **SOM_WarnMsg** macro prints the specified message, along with the filename and line number where the macro was invoked.

Related Information

Macros: SOM_Error

SOMClass Class

Description

SOMClass is the root class for all SOM metaclasses. That is, all SOM metaclasses must be subclasses of **SOMClass** or some other class derived from it. It defines the essential behavior common to all SOM classes. In particular, it provides a suite of methods for initializing class objects, generic methods for manufacturing instances of those classes, and methods that dynamically obtain or update information about a class and its methods at run time.

Just as all SOM classes are expected to have **SOMObject** (or a class derived from **SOMObject**) as their base class, all SOM classes are expected to have **SOMClass** or a class derived from **SOMClass** as their metaclass. Metaclasses define “class” methods (sometimes called “factory” methods or “constructors”) that manufacture objects from any class object that is defined as an instance of the metaclass.

To define your own class methods, define your own metaclass by subclassing **SOMClass** or one of its subclasses. Three methods that **SOMClass** inherits and overrides from **SOMObject** are typically overridden by any metaclass that introduces instance data—**somInit**, **somUninit**, and **somDumpSelfInt**. The new methods introduced in **SOMClass** that are frequently overridden are **somNew**, **somRenew**, and **somClassReady**. (See the descriptions of these methods for further information.)

Other reasons for creating a new metaclass include tracking object instances, automatic garbage collection, interfacing to a persistent object store, or providing/managing information that is global to a set of object instances.

File Stem

somcls

Base

SOMObject

Metaclass

SOMClass (**SOMClass** is the only class with itself as metaclass.)

Ancestor Classes

SOMObject

Types

```
typedef sequence <SOMClass> SOMClassSequence;

struct  somOffsetInfo  {
          SOMClass          cls;
          long              offset
          };
typedef sequence <somOffsetInfo> SOMOffsets;
```

New Methods

Attributes:

readonly attribute somOffsets somInstanceDataOffsets

_get_somInstanceDataOffsets returns a sequence of structures, each of which indicates an ancestor of the receiver class (or the receiver class itself) and the offset to the beginning of the instance data introduced by the indicated class in an instance of the receiver class. The **somOffsets** information can be used in conjunction with information derived from calls to a *SOM Interface Repository* to completely determine the layout of SOM objects at runtime.

C++ Example

```
#include <somcls.xh>
main()
{
   int i;
   SOMClassMgr *scm = somEnvironmentNew();
   somOffsets so = _SOMClass->_get_somInstanceDataOffsets();
   for (i=0; i<so._length; i++)
      printf(”In an instance of SOMClass, %s data starts at
%d\n”,
                 so._buffer[i]->cls->somGetName(),
                 so._buffer[i]->offset);
}
```

Introduced Methods

Group: Instance Creation (Factory)

somAllocate

somDeallocate

somNew, somNewNoInit

somRenew, somRenewNoInit, somRenewNoInitNoZero, somRenewNoZero

Group: Initialization/Termination

somAddDynamicMethod

somClassReady

Group: Access

somGetInstancePartSize

somGetInstanceSize

somGetInstanceToken

somGetMemberToken

somGetMethodData

somGetMethodDescriptor

somGetMethodIndex

somGetMethodToken

somGetName

somGetNthMethodData

somGetNthMethodInfo

somGetNumMethods

somGetNumStaticMethods

somGetParents

somGetVersionNumbers

Group: Testing

somCheckVersion

somDescendedFrom

somSupportsMethod

Group: Dynamic

somFindMethod, somFindMethodOk

somFindSMethod, somFindSMethodOk

somLookupMethod

Overridden Methods

somDefaultInit

somDestruct

somDumpSelfInt

Deprecated Methods

Use of the following methods is discouraged. There are three reasons for this:

First, these methods are used in constructing classes, and this capability is provided by the function **somBuildClass**. Class construction in SOM is currently a fairly complex activity, and it is likely to become even more so as the SOMobjects kernel evolves. To avoid breaking source code that constructs classes, you are advised to always use **somBuildClass** to build SOM classes.

Note: The SOM language bindings always use **somBuildClass**.

Second, these methods are used for customizing aspects of SOM classes, such as method resolution and object creation. Doing this requires that metaclasses override various methods introduced by **SOMClass**. However, if this is done without the Cooperation Framework that implements the SOM Metaclass Framework, SOMobjects cannot guarantee that applications will function correctly. Unfortunately, the Cooperation Framework (while available to SOM users as an experimental feature) is not officially supported by the SOMobjects Toolkit. So, this is another reason why the following methods are deprecated.

Finally, some of these methods are now obsolete, so it seems appropriate that their use be discouraged.

somAddStaticMethod

somGetApplyStub

somGetClassDatas

omGetClassMtab

somGetInstanceOffset

somGetMethodOffset

somGetParent

somGetPClsMtab

somGetPClsMtabs

somGetRdStub

somInitClass

somInitMIClass

somOverrideMtab

somOverrideSMethod

somSetClassData

somSetMethodDescriptor

somAddDynamicMethod Method

Purpose

Adds a new dynamic instance method to a class. Dynamic methods are not part of the declared interface to a class of objects, and are therefore not supported by implementation and usage bindings. Instead, dynamic methods provide a way to dynamically add new methods to a class of objects during execution. SOM provides no standard protocol for informing a user of the existence of dynamic methods and the arguments they take. Dynamic methods must be invoked using name-lookup or dispatch resolution.

IDL Syntax

void somAddDynamicMethod (
in somId *methodId***,**
in somId *methodDescriptor***,**
in somMethodPtr *method***,**
in somMethodPtr *applyStub***);**

Note: For backward compatibility, this method does *not* take an **Environment** parameter.

Description

The **somAddDynamicMethod** method adds a new dynamic instance method to the receiving class. This involves recording the method's ID, descriptor, method procedure (specified by *method*), and apply stub in the receiving class's method data.

The arguments to **somAddDynamicMethod** should be non-null and obey the following requirements. This is the responsibility of the implementor of a class, who in general has no knowledge of whether clients of this class will require use of the *applyStub* argument.

Parameters

receiver	A pointer to a SOM class object.
methodId	A **somId** that names the method.
methodDescriptor	A **somId** appropriate for requesting information concerning the method from the SOM IR. This is currently of the form <className>::<methodName>.
method	A pointer to the procedure that will implement the new method. The first argument of this procedure is the address of the object on which it is being invoked.
applyStub	A pointer to a procedure that returns nothing and receives as arguments: a method receiver; an address where the return value from the method call is to be stored; a pointer to a method procedure; and a va_list containing the arguments to the method. The applyStub procedure (which is usually called by **somDispatch**) must unload its va_list argument into separate variables of the correct type for the method, invoke its procedure argument on these variables, and then copy the result of the procedure invocation to the address specified by the return value argument.

C Example

```
/* New dynamic method ”newMethod1” for class ”XXX” */
static char *somMN_newMethod1 = ”newMethod1”;
static somId somId_newMethod1 = &somMN_newMethod1;
static char *somDS_newMethod1 = ”XXX::newMethod1”;
static somId somDI_newMethod1 = &somDS_newMethod1;

static void SOMLINK somAP_newMethod1(SOMObject somSelf,
                                     void *__retVal,
                                     somMethodProc *__methodPtr,
                                     va_list __ap)
{
   void* __somSelf = va_arg(__ap, SOMObject);
   int arg1 = va_arg(__ap, int);
   SOM_IgnoreWarning(__retVal);
   ((somTD_SOMObject_newMethod1) __methodPtr) (__somSelf, arg1);
}

main()
{
   _somAddDynamicMethod (
   XXXClassData.classObject,           /* Receiver (class object)
*/
   somId_newMethod1,                   /* method name somId
*/
   somDI_newMethod1,                   /* method descriptor somId
*/
   (somMethodProc *) newMethod1,       /* method procedure
*/
   (somMethodProc *) somAP_newMethod1); /* method apply stub
*/
}
```

Original Class

SOMClass

Related Information

Methods: somGetMethodDescriptor

somAllocate Method

Purpose

Supports class-specific memory allocation for class instances. Cannot be overridden.

IDL Syntax

string somAllocate (in long *size***);**

Note: For backward compatibility, this method does *not* take an **Environment** parameter.

Description

When building a class, the **somBuildClass** function is responsible for registering the procedure that will be executed when this method is invoked on the class. The default procedure registered by **somBuildClass** uses the **SOMMalloc** function, but the IDL modifier **somallocate** can be used in the SOM IDL class implementation section to indicate a different procedure. Users of this method should be sure to use the dual method, **somDeallocate**, to free allocated storage. Also, if the IDL modifier **somallocate** is used to indicate a special allocation routine, the IDL modifier **somdeallocate** should be used to indicate a dual procedure to be called when the **somDeallocate** method is invoked.

Parameters

receiver	A pointer to the class object whose memory allocation method is desired.
size	The number of bytes to be allocated.

Return Value

string	A pointer to the first byte of the allocated memory region, or NULL if sufficient memory is not available.

C++ Example

```
#include <som.xh>
#include <somcls.xh>
main()
{
   SOMClassMgr *cm = somEnvironmentNew();
   /* Use SOMClass's instance allocation method */
   string newRegion = _SOMClass->somAllocate(20);
}
```

Original Class

SOMClass

Related Information

Methods: somDeallocate

somCheckVersion Method

Purpose

Checks a class for compatibility with the specified major and minor version numbers. Not generally overridden.

IDL Syntax

boolean somCheckVersion (
In long *majorVersion*,
In long *minorVersion***);**

Note: For backward compatibility, this method does *not* take an **Environment** parameter.

Description

The **somCheckVersion** method checks the receiving class for compatibility with the specified major and minor version numbers. An implementation is compatible with the specified version numbers if it has the same major version number and a minor version number that is equal to or greater than *minorVersion*. The version number pair (0,0) is considered to match any version.

This method is called automatically after creating a class object to verify that a dynamically loaded class definition is compatible with a client application.

Parameters

receiver	A pointer to the SOM class whose version information should be checked.
majorVersion	This value usually changes only when a significant enhancement or incompatible change is made to a class.
minorVersion	This value changes whenever minor enhancements or fixes are made to a class. Class implementors usually maintain downward compatibility across changes in the *minorVersion* number.

Return Value

Returns 1 (true) if the implementation of this class is compatible with the specified major and minor version number, and 0 (false) otherwise.

C Example

```
#include <animal.h>
main()
{
  Animal myAnimal;
  myAnimal = AnimalNew();

  if (_somCheckVersion(_Animal, 0, 0))
     somPrintf(”Animal IS compatible with 0.0\n”);
  else
     somPrintf(”Animal IS NOT compatible with 0.0\n”);

  if (_somCheckVersion(_Animal, 1, 1))
     somPrintf(”Animal IS compatible with 1.1\n”);
  else
     somPrintf(”Animal IS NOT compatible with 1.1\n”);

  _somFree(myAnimal);
}
```

Assuming that the implementation of Animal is version 1.0, this program produces the following output:

```
Animal IS compatible with 0.0
Animal IS NOT compatible with 1.1
```

Original Class

SOMClass

somClassReady Method

Purpose

Indicates that a class has been constructed and is ready for normal use. Designed to be overridden.

IDL Syntax

void somClassReady ( );

Note: For backward compatibility, this method does *not* take an **Environment** parameter.

Description

The **somClassReady** method is invoked automatically by the **somBuildClass** function after constructing and initializing a class object. The default implementation of this method provided by **SOMClass** simply registers the newly constructed class with **SOMClassMgrObject**. Metaclasses can override this method to augment class construction with additional registration protocol.

To have special processing done when a class object is created, you must define a metaclass for the class that overrides **somClassReady**. The final statement in any overriding method should invoke the parent method to ensure that the class is properly registered with **SOMClassMgrObject**. Users of the C and C++ implementation bindings for SOM classes should never invoke the **somClassReady** method directly; it is invoked automatically during class construction.

Parameters

receiver	A pointer to the class object that should be registered.

Original Class

SOMClass

somDeallocate Method

Purpose

Frees memory originally allocated by the **somAllocate** method from the same class object. Cannot be overridden.

IDL Syntax

void somDeallocate (in string *memPtr***);**

Note: For backward compatibility, this method does *not* take an **Environment** parameter.

Description

The **somDeallocate** method is intended for use to free memory allocated using its dual method, **somAllocate**. When building a class, the **somBuildClass** function is responsible for registering the procedure that will be executed when this method is invoked on the class. The default procedure registered by **somBuildClass** uses the **SOMFree** function, but the IDL modifier **somdeallocate** can be used in the SOM IDL class implementation section to indicate a different procedure. Users of this method should be sure that the dual method, **somAllocate**, was originally used to allocate storage. Also, if the IDL modifier **somdeallocate** is used to indicate a special deallocation routine, the IDL modifier **somallocate** should be used to indicate a dual procedure to be called when **somAllocate** is invoked.

Parameters

receiver	A pointer to the class object whose **somAllocate** was originally used to allocate the memory now to be freed.
memPtr	A pointer to the first byte of the region of memory that is to be freed.

Original Class

SOMClass

Related Information

Methods: somAllocate

somDescendedFrom Method

Purpose

Tests whether one class is derived from another. Not generally overridden.

IDL Syntax

boolean somDescendedFrom (in SOMClass *aClassObj***);**

Note: For backward compatibility, this method does *not* take an **Environment** parameter.

Description

Tests whether the receiver class is derived from a given class. For programs that use classes as types, this method can be used to ascertain whether the type of one object is a subtype of another.

This method considers a class object to be descended from itself.

Parameters

receiver	A pointer to the class object to be tested.
aClassObj	A pointer to the potential ancestor class.

Return Value

Returns 1 (true) if *receiver* is derived from *aClassObj,* and 0 (false) otherwise.

C Example

```
#include <dog.h>
/* --------------------------------------------------------
   : Dog is a subclass of Animal.
   -------------------------------------------------------- */
main()
{
  AnimalNewClass(0,0);
  DogNewClass(0,0);

  if (_somDescendedFrom (_Dog, _Animal))
     somPrintf("Dog IS descended from Animal\n");
  else
     somPrintf("Dog is NOT descended from Animal\n");
  if (_somDescendedFrom (_Animal, _Dog))
     somPrintf("Animal IS descended from Dog\n");
  else
     somPrintf("Animal is NOT descended from Dog\n");
```

This program produces the following output:

```
Dog IS descended from Animal
Animal is NOT descended from Dog
```

Original Class

SOMClass

Related Information

Methods: somIsA, **somIsInstanceOf**

somFindMethod, somFindMethodOk Methods

Purpose

Finds the method procedure for a method and indicates whether it represents a static method or a dynamic method. Not generally overridden.

IDL Syntax

boolean somFindMethod (
in somId *methodId*,
out somMethodPtr *m*);

boolean somFindMethodOk (
in somId *methodId*,
out somMethodPtr *m*);

Note: For backward compatibility, these methods do *not* take an **Environment** parameter.

Description

The **somFindMethod** and **somFindMethodOk** methods perform name-lookup method resolution, determine the method procedure appropriate for performing the indicated method on instances of the receiving class, and load *m* with the method procedure address. For static methods, method procedure resolution is done using the instance method table of the receiving class.

Name-lookup resolution must be used to invoke dynamic methods. Also, name-lookup can be useful when different classes introduce methods of the same name, signature, and desired semantics, but it is not known until runtime which of these classes should be used as a type for the objects on which the method is to be invoked. If the signature of a method is an unknown, then method procedures cannot be be used directly, and the **somDispatch** method to be used after dynamically discovering the signature to allow the correct arguments can be placed on a va_list.

As with any methods that return procedure pointers, these methods allow repeated invocations of the same method procedure to be programmed. If this is done, it is up to the programmer to prevent runtime errors by assuring that each invocation is performed either on an instance of the class used to resolve the method procedure or of some class derived from it. Whenever using SOM method procedure pointers, it is necessary to indicate the arguments to be passed and the use of system linkage to the compiler, so it can generate a correct procedure call. The way this is done depends on the compiler and the system being used. However, C and C++ usage bindings provide an appropriate typedef for static methods. The name of the typedef is based on the name of the class that introduces the method, as illustrated in the following example.

Unlike the **somFindMethod** method, if the class does not support the specified method, the **somFindMethodOk** method raises an error and halts execution.

If the class does not support the specified method, then **m* is set to NULL and the return value is meaningless. Otherwise, the returned result is true if the indicated method was a static method.

Parameters

receiver	A pointer to the class object whose method is desired.
methodId	An ID that represents the name of the desired method. The **somIdFromString** function can used to obtain an ID from the method's name.
m	A pointer to the location in memory where a pointer to the specified method's procedure should be stored**.** Both methods store a NULL pointer in this location (if the method does not exist) or a value that can be called.

Return Value

The **somFindMethod** and **somFindMethodOk** methods return TRUE when the method procedure can be called directly and FALSE when the method procedure is a dispatch function.

C Example

Assuming that the *Animal* class introduces the method *setSound*, the type name for the *setSound* method procedure type will be *somTD_Animal_setSound*, as illustrated in the following example:

```
#include <animal.h>
void main()
{
   Animal myAnimal;
   somId somId_setSound;
   somTD_Animal_setSound methodPtr;
   Environment *ev = somGetGlobalEnvironment();

   myAnimal = AnimalNew();
/* ----------------------------------------
   : Next three lines are equivalent to
       _setSound(myAnimal, ev, ”Roar!!!”);
   ---------------------------------------- */
   somId_setSound = somIdFromString(”setSound”);
   _somFindMethod (_somGetClass(myAnimal),
                   somId_setSound, &methodPtr);
   methodPtr(myAnimal, ev, ”Roar!!!”);
/* ---------------------------------------- */
   _display(myAnimal, ev);
   _somFree(myAnimal);
}
/*
Program Output:
This Animal says
Roar!!!
*/
```

Original Class

SOMClass

Related Information

Methods: somFindSMethod, somFindSMethodOk, **somSupportsMethod**, **somDispatch, somClassDispatch**

Functions: somApply, **somResolve**, **somClassResolve**, **somResolveByName, somParentNumResolve**

Macros: SOM_Resolve, **SOM_ResolveNoCheck**, **SOM_ParentNumResolve**

somFindSMethod, somFindSMethodOk Methods

Purpose

Finds the method procedure for a static method. Not generally overridden.

IDL Syntax

somMethodPtr somFindSMethod (in somId *methodId***);**

somMethodPtr somFindSMethodOk (in somId *methodId***);**

Note: For backward compatibility, these methods do *not* take an **Environment** parameter.

Description

The **somFindSMethod** and **somFindSMethodOk** methods perform name-lookup resolution in a similar fashion to **somFindMethod** and **somFindMethodOk**, but are restricted to static methods. See the description of **somFindMethod** for a discussion of name-lookup method resolution. Because these methods are restricted to resolving static methods, their interface is slightly different from the **somFindMethod** interfaces; a method procedure pointer is returned when lookup is successful; otherwise NULL is returned.

The **somFindSMethodOk** method is identical to **somFindSMethod**, except that an error is raised if the indicated static method is not defined for the receiving class, and execution is halted.

Parameters

receiver	A pointer to a class object.
methodId	A somId representing the name of the desired method.

Return Value

The **somFindSMethod** and **somFindSMethodOk** methods return a pointer to the method procedure that supports the specified method for the class.

Example

See the **somFindMethod** example.

Original Class

SOMClass

Related Information

Methods: somFindMethod, somFindMethodOk Methods

somGetInstancePartSize Method

Purpose

Returns the total size of the instance data structure introduced by a class. Not generally overridden.

IDL Syntax

long somGetInstancePartSize ( );

Note: For backward compatibility, this method does *not* take an **Environment** parameter.

Description

The **somGetInstancePartSize** method returns the amount of space needed in an object of the specified class or any of its subclasses to contain the instance variables introduced by the class.

Parameters

receiver	A pointer to the class object whose instance data size is desired.

Return Value

The **somGetInstancePartSize** method returns the size, in bytes, of the instance variables introduced by this class. This does not include the size of instance variables introduced by this class's ancestor or descendent classes. If a class introduces no instance variables, 0 is returned.

C Example

```
#include <animal.h>
main()
{
  Animal myAnimal;
  SOMClass animalClass;
  int instanceSize;
  int instanceOffset;
  int instancePartSize;

  myAnimal = AnimalNew ();
  animalClass = _somGetClass (myAnimal);
  instanceSize = _somGetInstanceSize (animalClass);
  instanceOffset = _somGetInstanceOffset (animalClass);
  instancePartSize = _somGetInstancePartSize (animalClass);
  somPrintf (”Instance Size: %d\n”, instanceSize);
  somPrintf (”Instance Offset: %d\n”, instanceOffset);
  somPrintf (”Instance Part Size: %d\n”, instancePartSize);
  _somFree (myAnimal);
}
/*
Output from this program:
Instance Size: 8
Instance Offset: 0
Instance Part Size: 4
*/
```

Original Class

SOMClass

Related Information

Methods: somGetInstanceSize

somGetInstanceSize Method

Purpose

Returns the size of an instance of a class. Not generally overridden.

IDL Syntax

long somGetInstanceSize ( );

Note: For backward compatibility, this method does *not* take an **Environment** parameter.

Description

The **somGetInstanceSize** method returns the total amount of space needed in an instance of the specified class.

Parameters

receiver	A pointer to the class object whose instance size is desired.

Return Value

The **somGetInstanceSize** method returns the size, in bytes, of each instance of this class. This includes the space required for instance variables introduced by this class and all of its ancestor classes.

C Example

```
#include <animal.h>
main()
{
  Animal myAnimal;
  SOMClass animalClass;
  int instanceSize;
  int instanceOffset;
  int instancePartSize;

  myAnimal = AnimalNew ();
  animalClass = _somGetClass (myAnimal);
  instanceSize = _somGetInstanceSize (animalClass);
  instanceOffset = _somGetInstanceOffset (animalClass);
  instancePartSize = _somGetInstancePartSize (animalClass);
  somPrintf (”Instance Size: %d\n”, instanceSize);
  somPrintf (”Instance Offset: %d\n”, instanceOffset);
  somPrintf (”Instance Part Size: %d\n”, instancePartSize);
  _somFree (myAnimal);
}
/*
Output from this program:
Instance Size: 8
Instance Offset: 0
Instance Part Size: 4
*/
```

Original Class

SOMClass

Related Information

Methods: somGetInstancePartSize

somGetInstanceToken Method

Purpose

Returns a data access token for the instance data introduced by a class. Not generally overridden.

IDL Syntax

somDToken somGetInstanceToken ( );

Note: For backward compatibility, this method does *not* take an **Environment** parameter.

Description

Returns a data token "pointing" to the beginning of the instance data introduced by the receiving class. This token can be passed to the function **somDataResolve** to locate this instance data within an an instance of the receiver class or any class derived from it. Also the instance data token for a class can be passed to the class method **somGetMemberToken** to get a data token for a specific instance variables introduced by the class (if the relative offset of this instance variable is known). This approach is used by C and C++ implementation bindings to support public instance data for OIDL classes (IDL classes currently have no public instance data).

A data token for the instance data introduced by a class is required by method procedures that access data introduced by the method procedure's defining class. For classes declared using OIDL and IDL, the needed token is stored in the auxiliary class data structure, which is an external data structure made statically available by the C and C++ language bindings as *<className>*CClassData.instanceToken. Thus, this method call is not generally used by C and C++ class implementors of classes declared using OIDL or IDL.

Parameters

receiver	A pointer to a **SOMClass** object.

Return Value

Returns a data token for the beginning of the instance data introduced by the receiver.

Original Class

SOMClass

Related Information

Methods: somGetInstanceSize, **somGetInstancePartSize**, **somGetMemberToken**

Functions: somDataResolve

somGetMemberToken Method

Purpose

Returns an access token for an instance variable. This is method is not generally overridden.

IDL Syntax

somDToken somGetMemberToken (
long *memberOffset***,**
somDToken *instanceToken***);**

Note: For backward compatibility, this method does *not* take an **Environment** parameter.

Description

The **somGetMemberToken** method returns an access token for the data member at offset *memberOffset* within the block of instance data identified by *instanceToken*. The returned token can subsequently be passed to the **somDataResolve** function to locate the data member.

Typically, only the code that implements a class declared using OIDL requires access to this method, and this code is normally provided by implementation bindings. Thus C and C++ programmers do not normally invoke this method.

Parameters

receiver	A pointer to a **SOMClass** object.
memberOffset	A 32-bit integer representing the offset of the required data member.
instanceToken	A token, obtained from **somGetInstanceToken**, that identifies the introduced portion of the class.

Return Value

Returns an access token for the specified data member.

Original Class

SOMClass

Related Information

Methods: somGetInstanceSize, **somGetInstancePartSize**, **somGetInstanceToken**

Functions: somDataResolve

somGetMethodData Method

Purpose

Returns method information for a specified method, which must have been introduced by the receiver class or an ancestor of that class. Not generally overridden.

IDL Syntax

boolean somGetMethodData (
in somId *methodId***,**
out somMethodData *md***);**

Note: For backward compatibility, this method does *not* take an **Environment** parameter.

Description

The **somGetMethodData** method loads a *somMethodData* structure with data describing the method identified by the passed *methodId*. If *methodId* does not identify a method known to the receiver, then false is returned; otherwise, true is returned after loading the *somMethodData* structure with data corresponding to the indicated method.

Parameters

receiver	A pointer to the class that produced the index value.
methodId	A **somId** for the method's name.
md	A pointer to a *somMethodData* structure.

Return Value

Boolean true if successful; otherwise false.

C++ Example

```
#include <somcls.xh>
main
{
   somEnvironmentNew();
   somId gmiId = somIdFromString(”somGetMethodIndex”);
   somMethodData md;
   boolean rc = _SOMClass->somGetMethodData(gmiId,&md);
   SOM_Test(rc && somCompareIds(gmiId, md.id));
}
```

Related Information

Methods: somGetMethodIndex, **somGetMethodData**, **somGetNthMethodInfo**

Data Structures: somMethodData (**somapi.h**)

somGetMethodDescriptor Method

Purpose

Returns the method descriptor for a method. Not generally overridden.

IDL Syntax

somId somGetMethodDescriptor (in somId *methodId***);**

Note: For backward compatibility, this method does *not* take an **Environment** parameter.

Description

The **somGetMethodDescriptor** method returns the method descriptor for a specified method of a class. (A method descriptor is a somId that represents the identifier of an attribute definition or a method definition in the SOM Interface Repository. It contains information about the method's return type and the types of its arguments.) If the class object does not support the indicated method, NULL is returned.

Parameters

receiver	A pointer to a **SOMClass** object.
methodId	A **somId** method descriptor.

Return Value

The **somGetMethodDescriptor** method returns a **somId** method descriptor.

Example

```
somId myMethodDescriptor;
myMethodDescriptor = _somGetMethodDescriptor(_Animal,
                               somIdFromString("setSound"));
```

Original Class

SOMClass

Related Information

Methods: somAddDynamicMethod, **somGetNthMethodInfo**, **somGetMethodData**, **somGetNthMethodData**

somGetMethodIndex Method

Purpose

Returns a class-specific index for a method. Not generally overridden.

IDL Syntax

long somGetMethodIndex (in somId *methodId***);**

Note: For backward compatibility, this method does *not* take an **Environment** parameter.

Description

The **somGetMethodIndex** method returns an index that can be used in subsequent calls to the same receiving class to determine information about the indicated (static or dynamic) method, via the methods **somGetNthMethodData** and **somGetNthMethodInfo**. The method must be appropriate for use on an instance of the receiver class; otherwise, a –1 is returned. The index of a method can change over time if dynamic methods are added to the receiver class or its ancestors. Thus, in dynamic multi-threaded environments, a critical region should be used to bracket the use of this method and of subsequent requests for method information based on the returned index.

Parameters

receiver	A pointer to a **SOMClass** object.
methodId	A **somId** method ID.

Return Value

The **somGetMethodIndex** method returns a positive long if successful, and a –1 otherwise.

C++ Example

```
#include <somcls.xh>
main
{
   somEnvironmentNew();
   somId gmiId = somIdFromString(”somGetMethodIndex”);
   long index = _SOMClass->somGetMethodIndex(gmiId);
   somMethodData md;
   boolean rc = _SOMClass->somGetNthMethodData(index,&md);
   SOM_Test(rc && somCompareIds(gmiId, md.id));
}
```

Original Class

SOMClass

Related Information

Methods: somGetNthMethodData, **somGetNthMethodInfo**

Data Structures: somMethodData (**somapi.h**)

somGetMethodToken Method

Purpose

Returns a method access token for a static method. Not generally overridden.

IDL Syntax

somMToken somGetMethodToken (in somId *methodId***);**

Note: For backward compatibility, this method does *not* take an **Environment** parameter.

Description

The **somGetMethodToken** method returns a method access token for a static method with the specified ID that was introduced by the receiver class or an ancestor of the receiver class. This method token can be passed to the **somResolve** function (or one of the other offset-based method resolution functions) to select a method procedure pointer from a method table of an object whose class is the same as, or is derived from the class that introduced the method.

Parameters

receiver	A pointer to a **SOMClass** object.
methodId	A **somId** identifying a method.

Return Value

The **somGetMethodToken** method returns a **somMToken** method-access token.

C Example

Assuming that the class *Animal* introduces the method *setSound*,

```
#include <animal.h>
main() {
  somMToken tok;
  Animal myAnimal;
  somTD_Animal_setSound methodPtr; /* use typedef from animal.h
*/
  Environment *ev = somGetGlobalEnvironment();
  myAnimal = AnimalNew();
  /*next 3 lines equivalent to _setSound(myAnimal, ev,
"Roar!!!");*/
  tok = _somGetMethodToken(_Animal, somIdFromString("setSound"));
  methodPtr = (somTD_Animal_setSound)somResolve(myAnimal, tok);
  methodPtr(myAnimal, ev, "Roar!!!");
  _display(myAnimal, ev);
  _somFree(myAnimal);
}
```

Original Class

SOMClass

Related Information

Methods: somGetNthMethodInfo, **somGetMethodData**

Functions: somResolve, **somClassResolve**, **somParentNumResolve**

somGetName Method

Purpose

Returns the name of a class. Not generally overridden.

IDL Syntax

string somGetName ( );

Note: For backward compatibility, this method does *not* take an **Environment** parameter.

Description

The **somGetName** method returns the address of a zero-terminated string that gives the name of the receiving class. This name may be used as a RepositoryId in the **Repository_lookup_id** method (described in the SOM Interface Repository Framework section) to obtain the IDL interface definition that corresponds to the receiving class.

The returned name is not necessarily the same as the statically known class name used by a programmer to gain access to the class object (for example, via the method **somFindClass**). This is because the method **somSubstituteClass** may have been used to "shadow" the class having the static name used by the programmer.

Also, when the interface to a class's instances is defined within an IDL module, the returned name will not directly correspond to the names of the procedures and macros made available by the SOMobjects C and C++ usage bindings for accessing class objects (for example, the **<***className***>NewClass** procedure, or the **_<***className***>** macro). This is because the <*className*> token used in constructing the names of these procedures and macros uses an underscore character to separate the module name from the interface name, while the actual name of the corresponding class uses two colon characters instead of an underscore for this purpose.

The **somGetName** method is not generally overridden. The returned address is valid until the class object is unregistered or freed.

Parameters

receiver The class whose name is desired.

Return Value

The **somGetName** method returns a pointer to the name of the class.

C++ Example

```
#include <animal.xh> /* assume Animal defined in the Zoo module
*/
#include <string.h>
main()
{
  string className = Zoo_AnimalNewClass(0,0)->somGetName();
  SOM_Test(!strcmp(className, "Zoo::Animal"));
}
```

Original Class

SOMClass

Related Information

Methods: Repository_lookup_id, **somSubstituteClass, somFindClass**

somGetNthMethodData Method

Purpose

Returns method information for the *n*th (static or dynamic) method known to a given class. Not generally overridden.

IDL Syntax

boolean somGetNthMethodData (
in long *index*,
out somMethodData *md***)**

Note: For backward compatibility, this method does *not* take an **Environment** parameter.

Description

The **somGetNthMethodData** method loads a *somMethodData* structure with data describing the method identified by the passed index. The index must have been produced by a previous call to exactly the same receiver class; the same method will in general have different indexes in different classes. If the index does not identify a method known to this class, then false is returned; otherwise, true is returned after loading the *somMethodData* structure with data corresponding to the indicated method.

Parameters

receiver	A pointer to the class that produced the index value.
index	An index returned as a result of a previous call of **somGetMethodIndex**.
md	A pointer to a *somMethodData* structure.

Return Value

Boolean true if successful; otherwise, false.

C++ Example

```
#include <somcls.xh>
main
{
   somEnvironmentNew();
   somId gmiId = somIdFromString(”somGetMethodIndex”);
   long index = _SOMClass->somGetMethodIndex(gmiId);
   somMethodData md;
   boolean rc = _SOMClass->somGetNthMethodData(index,&md);
   SOM_Test(rc && somCompareIds(gmiId, md.id));
}
```

Related Information

Methods: somGetMethodIndex, **somGetMethodData**, **somGetNthMethodInfo**

Data Structures: somMethodData (**somapi.h**)

somGetNthMethodInfo Method

Purpose

Returns the **somId** of the *n*th (static or dynamic) method known to a given class. Also loads a **somId** with a descriptor for the method. Not generally overridden.

IDL Syntax

somId somGetNthMethodInfo (
in long *index***,**
out somId *descriptor***);**

Note: For backward compatibility, this method does *not* take an **Environment** parameter.

Description

The **somGetNthMethodInfo** method returns the identifier of a method, and loads the **somId** whose address is passed with the **somId** of the method descriptor. Method descriptors are used to support access to information stored in a SOM Interface Repository.

Parameters

receiver	A pointer to the class from which the *index* was obtained using method **somGetMethodIndex**.
index	The *n*th method known to this class, whose method descriptor is desired.
descriptor	A pointer to a **somId** that will be loaded with a **somId** for the descriptor.

Return Value

The **somId** for the indicated method, if a method with the indicated index is known to the receiver; otherwise, NULL.

C++ Example

```
#include <somcls.xh>
main()
{
   somEnvironmentNew();
   somId descriptor, icId = somIdFromString(”somInitClass”);
   long ndx = _SOMClass->somGetMethodIndex(icId);
   SOM_Test(
      somCompareIds(
        icId,
        _SOMClass->somGetNthMethodInfo(ndx,&descriptor));
   SOMFree(icId);
   SOMFree(descriptor);
}
```

Original Class

SOMClass

Related Information

Methods: somGetMethodIndex, **somGetNthMethodData**

Classes: Repository (**repostry.idl**)

somGetNumMethods Method

Purpose

Returns the number of methods available for a class. Not generally overridden.

IDL Syntax

long somGetNumMethods ( );

Note: For backward compatibility, this method does *not* take an **Environment** parameter.

Description

The **somGetNumMethods** method returns the number of methods currently supported by the specified class, including inherited methods (both static and dynamic).

The value that the **somGetNumMethods** method returns is the total number of methods currently known to the receiving class as being applicable to its instances. This includes both static and dynamic methods, whether defined in this class or inherited from an ancestor class.

Parameters

receiver	A pointer to the class whose instance method count is desired.

Return Value

The **somGetNumMethods** method returns the total number of methods that are currently available for the receiving class.

C Example

```
#include <animal.h>
main()
{
    int numMethods;

    AnimalNewClass(Animal_MajorVersion, Animal_MinorVersion);
    numMethods = _somGetNumMethods(_Animal);
    somPrintf(”Number of methods supported by class: %d\n”,
                                              numMethods);
}
```

Original Class

SOMClass

Related Information

Methods: somGetNumStaticMethods

somGetNumStaticMethods Method

Purpose

Obtains the number of static methods available for a class. Not generally overridden.

IDL Syntax

long somGetNumStaticMethods ( );

Note: For backward compatibility, this method does *not* take an **Environment** parameter.

Description

The **somGetNumStaticMethods** method returns the number of static methods available in the specified class, including inherited ones. Static methods are those that are represented by entries in the class's instance method table, and which can be invoked using method tokens and offset resolution.

Parameters

receiver	A pointer to the class whose static method count is desired.

Return Value

The **somGetNumStaticMethods** method returns the total number of static methods that are available for instances of the receiving class.

C Example

```
#include <animal.h>
main()
{
    int numMethods;

    AnimalNewClass(Animal_MajorVersion, Animal_MinorVersion);
    numMethods = _somGetNumStaticMethods(_Animal);
    somPrintf("Number of static methods supported by class:
%d\n",
                 numMethods);
}
```

Original Class

SOMClass

Related Information

Methods: somGetNumMethods

somGetParents Method

Purpose

Gets a pointer to a class's parent (direct base) classes. Not generally overridden.

IDL Syntax

SOMClassSequence somGetParents ( );

Note: For backward compatibility, this method does *not* take an **Environment** parameter.

Description

The **somGetParents** method returns a sequence containing pointers to the parents of the receiver.

Parameters

receiver	A pointer to the class whose parent (base) classes are desired.

Return Value

The **somGetParents** method returns a sequence of pointers to the parents of the receiver, or NULL otherwise (in the case of **SOMObject**). The sequence of parents is in left-to-right order.

C Example

```
/* : Dog is a single-inheritance subclass of Animal. */
#include <dog.h>
main()
{
  Dog myDog;
  SOMClass dogClass;
  SOMClassSequence parents;
  char *parentName;
  int i;

  myDog = DogNew();
  dogClass = _somGetClass(myDog);
  parents = _somGetParents(dogClass);
  for (i=0; i<parents._length; i++)
     somPrintf("-- parent %d is %s\n", i,
               _somGetName(parents._buffer[i]));
  _somFree(myDog);
}
/*
Output from this program:
-- parent 0 is Animal
*/
```

Original Class

SOMClass

Related Information

Methods: somGetClass

somGetVersionNumbers Method

Purpose

Gets the major and minor version numbers of a class’s implementation code. Not generally overridden.

IDL Syntax

void somGetVersionNumbers (
out long *majorVersion*,
out long *minorVersion***);**

Note: For backward compatibility, this method does *not* take an **Environment** parameter.

Description

The **somGetVersionNumbers** method returns, via its output parameters, the major and minor version numbers of the class specified by *receiver*. The class object must have already been created (because the class object is the receiver of the method).

Parameters

receiver	A pointer to a class object.
majorVersion	A pointer where the major version number is to be stored.
minorVersion	A pointer where the minor version number is to be stored.

C Example

```
#include <som.h>

main() {

  long major, minor;
  SOMClass myClass;

  somEnvironmentNew();
  myClass = _somFindClass(SOMClassMgrObject,
                          somIdFromString(”Animal”), 0, 0);
  _somGetVersionNumbers(myClass, &major, &minor);
  somPrintf(”The version numbers are %i and %i.\n”, major,
minor);
}
```

Original Class

SOMClass

Related Information

Methods: somCheckVersion

somLookupMethod Method

Purpose

Performs name-lookup method resolution. Not generally overridden.

IDL Syntax

somMethodPtr somLookupMethod (in somId *methodId***);**

Note: For backward compatibility, this method does *not* take an **Environment** parameter.

Description

The **somLookupMethod** method uses name-lookup resolution to return the address of the method procedure that supports the indicated method on instances of the receiver class. The method may be either static or dynamic. If the method is not supported by the receiving class, then NULL is returned. The SOM C and C++ usage bindings support name-lookup method resolution by invoking **somLookupMethod** on the class of the object on which a name-lookup method invocation is made.

The **somLookupMethod** method is like **somFindSMethod** except that dynamic methods can also be returned.

As always, in order to use a method procedure pointer such as that returned by **somLookupMethod**, it is necessary to typecast the procedure pointer so that the compiler can create the correct procedure call. This means that a programmer making explicit use of this method must either know the signature of the identified method, and from this create a typedef indicating system linkage and the appropriate argument and return types, or make use of an existing typedef provided by C or C++ usage bindings for a SOM class that introduces a static method with the desired signature.

Parameters

receiver	A pointer to the class whose instance method for the indicated method is desired.
methodId	A **somId** of the method whose method-procedure pointer is needed.

Return Value

A pointer to the method procedure that supports the method indicated by *methodId.*

C++ Example

```
#include <somcls.xh>
#include <somcm.xh>
void main()
{
   somId fcpId = somIdFromString(”somFindClass”)
   somId animalId = somIdFromString(”Animal”);
   SOMClassMgr *cm = somEnvironmentNew();
   somTD_SOMClassMgr_somFindClass findclassproc =
         (somTD_SOMClassMgr_somFindClass)
             _SOMClassMgr->somLookupMethod(fcpId);
   SOMClass *aCls = findclassproc(cm,animalId,0,0);
   ...
   somFree(fcpId);
   somFree(animalId);
 }
```

Original Class

SOMClass

Related Information

Methods: somFindSMethod, somFindSMethodOk Methods, **somFindMethod, somFindMethodOk Methods**

somNew, somNewNoInit Methods

Purpose

Creates a new instance of a class.

IDL Syntax

SOMObject somNew ( );

SOMObject somNewNoInit ( );

Note: For backward compatibility, these methods do *not* take an **Environment** parameter.

Description

The **somNew** and **somNewNoInit** methods create a new instance of the receiving class. Space is allocated as necessary to hold the new object.

When either of these methods is applied to a class, the result is a new instance of that class. If the receiver class is **SOMClass** or a class derived from **SOMClass**, the new object will be a class object; otherwise, the new object will not be a class object. The **somNew** method invokes the **somDefaultInit** method on the newly created object. The **somNewNoInit** method does not.

Either method can fail to allocate enough memory to hold a new object and, if so, NULL is returned.

The SOM Compiler generates convenience macros for creating instances of each class, for use by C and C++ programmers. These macros can be used in place of this method.

Parameters

receiver	A pointer to the class object that is to create a new instance.

Return Value

A pointer to the newly created **SOMObject** object, or NULL.

Example

```
#include <animal.h>

void main()
{    Animal myAnimal;
/* -------------------------------------------------
: next 2 lines are functionally equivalent to
        myAnimal = AnimalNew();
------------------------------------------------- */
    /* Create class object:. */
    AnimalNewClass(Animal_MajorVersion, AnimalMinorVersion);
    myAnimal = _somNew(_Animal);  /* Create instance of Animal
cls */
    /* ... */
    _somFree(myAnimal);       /* Free instance of Animal */
 }
```

Original Class

SOMClass

Related Information

Methods: somRenew

somRenew, somRenewNoInit, somRenewNoInitNoZero, somRenewNoZero Methods

Purpose

Creates a new object instance using a passed block of storage.

IDL Syntax

SOMObject somRenew (in somToken *memPtr***);**

SOMObject somRenewNoInit (in somToken *memPtr***);**

SOMObject somRenewNoInitNoZero (in somToken *memPtr***);**

SOMObject somRenewNoZero (in somToken *memPtr***);**

Note: For backward compatibility, these methods do *not* take an **Environment** parameter.

Description

The **somRenew** method creates a new instance of the receiving class by setting the appropriate location in the passed memory block to the receiving class's instance method table. Unlike **somNew**, these "Renew" methods use the space pointed to by *memPtr* rather than allocating new space for the object. The **somRenew** method automatically re-initializes the object by first zeroing the object's memory, and then invoking **somInit**; **somRenewNoInit** zeros memory, but does not invoke **somInit**. **somRenewNoInitNoZero** only sets the method table pointer; while **somRenewNoZero** calls **somInit**, but does not zero memory first.

No check is made to ensure that the passed pointer addresses enough space to hold an instance of the receiving class. The caller can determine the amount of space necessary by using the **somGetInstanceSize** method.

The C bindings produced by the SOM Compiler contain a macro that is a convenient shorthand for **_somRenew(_***className***)**.

Parameters

receiver	A pointer to the class object that is to create the new instance.
memPtr	A pointer to the space to be used to construct a new object.

Return Value

The value of *newObject* is returned, which is now a pointer to a valid, initialized object.

Example

```
#include <animal.h>

main()
{
  void *myAnimalCluster;
  Animal animals[5];
  SOMClass animalClass;
  int animalSize, i;

  animalClass =
      AnimalNewClass(Animal_MajorVersion,Animal_MinorVersion);
  animalSize = _somGetInstanceSize (animalClass);
  /* Round up to double-word multiple */
  animalSize = ((animalSize+3)/4)*4;
  /*
   * Next line allocates room for 5 objects
   * in a &odq.cluster” with a single memory-
   * allocation operation.
   */
  myAnimalCluster = SOMMalloc (5*animalSize);
  /*
   * The for-loop that follows creates 5 initialized
   * Animal instances within the memory cluster.
   */
  for (i=0; i<5; i++)
     animals[i] =
      _somRenew(animalClass, myAnimalCluster+(i*animalSize));
  /* Uninitialize the animals explicitly: */
  for (i=0; i<5; i++)
     _somUninit(animals[i]);
  /*
   * Finally, the next line frees all 5 animals
   * with one operation.
   */
  SOMFree (myAnimalCluster);
}
```

Original Class

SOMClass

Related Information

Methods: somGetInstanceSize, **somInit**, **somNew**

somSupportsMethod Method

Purpose

Returns a **boolean** indicating whether instances of a class support a given (static or dynamic) method.

IDL Syntax

boolean somSupportsMethod (in somId *methodId***);**

Note: For backward compatibility, this method does *not* take an **Environment** parameter.

Description

The **somSupportsMethod** method determines if instances of the specified class support the specified (static or dynamic) method.

Parameters

receiver	A pointer to the class object to be tested.
methodId	An ID that represents the name of the method.

Return Value

The **somSupportsMethod** method returns 1 (true) if instances of the specified class support the specified method, and 0 (false) otherwise.

Example

```
/* ----------------------------------------------
   : animal supports a setSound method;
         animal does not support a doTrick method.
   ---------------------------------------------- */
#include <animal.h>
main()
{
  SOMClass animalClass;
  char *methodName1 = ”setSound”;
  char *methodName2 = ”doTrick”;
  animalClass =
      AnimalNewClass(Animal_MajorVersion, Animal_MinorVersion);
  if (_somSupportsMethod(animalClass,
                         somIdFromString(methodName1)))
     somPrintf(”Animals respond to %s\n”, methodName1);
  if (_somSupportsMethod(animalClass,
                         somIdFromString(methodName2)))
     somPrintf(”Animals respond to %s\n”, methodName2);
}

/*
Output from this program:
Animals respond to setSound
*/
```

Original Class

SOMClass

Related Information

Methods: somRespondsTo

SOMClassMgr Class

Description

One instance of **SOMClassMgr** is created automatically during SOM initialization. This instance (pointed to by the global variable, **SOMClassMgrObject**) acts as a run-time registry for all SOM class objects that exist within the current process and assists in the dynamic loading and unloading of class libraries.

You can subclass **SOMClassMgr** to augment the functionality of its registry. To have an instance of your subclass replace the SOM-supplied **SOMClassMgrObject**, use the **somMergeInto** method to place the existing registry information from **SOMClassMgrObject** into your new class-manager object.

File Stem

somcm

Base

SOMObject

Metaclass

SOMClass

Ancestor Classes

SOMObject

Types

interface Repository;
SOMClass *SOMClassArray;

Attributes

The following is a list of each available attribute with its corresponding type in parentheses, followed by a description of its purpose.

somInterfaceRepository (Repository)
The SOM Interface Repository object. If the Interface Repository is not available or cannot be initialized, this attribute returns NULL. The object reference returned by this attribute is owned by the **SOMClassMgr** and should not be freed.

somRegisteredClasses (sequence<SOMClass>)
This is a “readonly” attribute that returns a sequence containing all of the class objects registered in the current process. When you have finished using the returned sequence, you should free the sequence’s buffer using **SOMFree**. Here is a fragment of code written in C that illustrates the proper use of this attribute:

```
sequence(SOMClass) clsList;

clsList = SOMClassMgr__get_somRegisteredClasses
(SOMClassMgrObject);
somPrintf (”Currently registered classes:\n”);
for (i=0; i<clsList._length; i++)
    somPrintf (”\t%s\n”, SOMClass_somGetName
(clsList._buffer[i]));
SOMFree (clsList._buffer);
```

New Methods

Group: Basic Functions

somLoadClassFile

somLocateClassFile

somRegisterClass

somUnloadClassFile

somUnregisterClass

Group: Access

somGetInitFunction

somGetRelatedClasses

Group: Dynamic

somClassFromId

somFindClass

somFindClsInFile

somMergeInto

somSubstituteClass

Overridden Methods

somDumpSelf

somInit

somUninit

somClassFromId Method

Purpose

Finds a class object, given its somId, if it already exists. Does not load the class.

IDL Syntax

SOMClass somClassFromId (in somId *classId***);**

Note: For backward compatibility, this method does *not* take an **Environment** parameter.

Description

Finds a class object, given its somId, if it already exists. Does not load the class.

Use the **somClassFromId** method instead of **somFindClass** when you do *not* want the class to be automatically loaded if it does not already exist in the current process.

Parameters

receiver	Usually **SOMClassMgrObject** (or a pointer to an instance of a user-supplied subclass of **SOMClassMgr**).
classId	The somId of the class. This can be obtained from the name of the class using the **somIdFromString** function.

Return Value

Returns a pointer to the class, or NULL if the class object does not yet exist.

C Example

```
#include <som.h>

main () {
    SOMClass myClass;
    char *myClassName = ”Animal”;
    somId animalId;

    somEnvironmentNew ();
    animalId = somIdFromString (myClassName);
    myClass = SOMClassMgr_somClassFromId (SOMClassMgrObject,
                                                         animalId);
    if (!myClass)
        somPrintf (”Class %s has not been loaded.\n”, myClassName);
    SOMFree (animalId);
    }
```

This program produces the following output:

```
  Class Animal has not yet been loaded.
```

Original Class

SOMClassMgr

Related Information

Methods: somFindClass, **somFindClsInFile**

somFindClass Method

Purpose

Finds the class object for a class.

IDL Syntax

```
SOMClass  somFindClass (
                         in somId classId,
                         in long majorVersion,
                         in long minorVersion);
```

Note: For backward compatibility, this method does *not* take an **Environment** parameter.

Description

The **somFindClass** method returns the class object for the specified class. This method first uses **somLocateClassFile** (see the following paragraph) to obtain the name of the file where the class's code resides, then uses **somFindClsInFile**.

If the requested class has not yet been created, the **somFindClass** method attempts to load the class dynamically by loading its dynamically linked library and invoking its "new class" procedure.

The **somLocateClassFile** method uses the following steps:

1. If the entry in the Interface Repository for the class specified by *classId* contains a **dllname** modifier, this value is used as the file name for loading the library. (For information about the **dllname** modifier, refer to the topic "Modifier statements" in Chapter 4, "SOM IDL and the SOM Compiler," of the *SOMobjects Developer Toolkit Users Guide.*)
2. In the absence of a **dllname** modifier, the class name is assumed to be the file name for the library. Use the **somFindClsInFile** method if you wish to explicitly pass the file name as an argument.

If *majorVersion* and *minorVersion* are not both zero, they are used to check the class version information against the caller's expectations. An implementation is compatible with the specified version numbers if it has the same major version number and a minor version number that is equal to or greater than *minorVersion*.

Parameters

receiver	Usually **SOMClassMgrObject** (or a pointer to an instance of a user-supplied subclass of **SOMClassMgr**).
classId	The **somId** representing the name of the class.
majorVersion	The class's major version number.
minorVersion	The class's minor version number.

Return Values

A pointer to the requested class object, or NULL if the class could not be found or created.

C Example

```
#include <som.h>

/*
 *  This program creates a class object
 *  (from a DLL) without requiring the
 *  usage binding file (.h or .xh) for
 *  the class.
 */

void main ()
{
    SOMClass myClass;
    somId animalId;

    somEnvironmentNew ();
    animalId = somIdFromString (”Animal”);

/*  The next statement is equivalent to:
 *    #include ”animal.h”
 *    myClass = AnimalNewClass (0, 0);
 */
    myClass = SOMClassMgr_somFindClass (SOMClassMgrObject,
                                            animalId, 0, 0);
    if (myClass)
        somPrintf (”myClass: %s\n”, SOMClass_somGetName
(myClass));
    else
        somPrintf (”Class %s could not be dynamically loaded\n”,
                                       somStringFromId
(animalId));
    SOMFree (animalId);
}
```

This program produces the following output:

```
myClass: Animal
```

Original Class

SOMClassMgr

Related Information

Methods: somFindClsInFile, **somLocateClassFile**

somFindClsInFile Method

Purpose

Finds the class object for a class, given a filename that can be used for dynamic loading.

IDL Syntax

```
SOMClass  somFindClsInFile (
                                in somId classId,
                                in long majorVersion,
                                in long minorVersion,
                                in string file);
```

Note: For backward compatibility, this method does *not* take an **Environment** parameter.

Description

The **somFindClsInFile** method returns the class object for the specified class. This method is the same as **somFindClass** except that the caller provides the filename to be used if dynamic loading is needed.

If the requested class has not yet been created, the **somFindClsInFile** method attempts to load the class dynamically by loading the specified library and invoking its “new class” procedure.

If *majorVersion* and *minorVersion* are not both zero, they are used to check the class version information against the caller’s expectations. An implementation is compatible with the specified version numbers if it has the same major version number and a minor version number that is equal to or greater than *minorVersion*.

Parameters

receiver	Usually **SOMClassMgrObject** (or a pointer to an instance of a user-supplied subclass of **SOMClassMgr**).
classId	The somId representing the name of the class.
majorVersion	The class’s major version number.
minorVersion	The class’s minor version number.
file	A string representing the filename to be used if dynamic loading is required.

Return Value

A pointer to the requested class object, or NULL if the class could not be found or created.

C Example

```
#include <som.h>
/*
 *  This program loads a class and creates
 *  an instance of it without requiring the
 *  binding (.h) file for the class.
 *
 */
void main()
{
   SOMObject myAnimal;
   SOMClass animalClass;
   char *animalName = ”Animal”;
        /*
         * Filenames will be different for AIX, OS/2 and Windows
         *
         * Set animalfile to ”C:\\MYDLLS\\ANIMAL.DLL” for OS/2
         *                                           or Windows.
         * Set animalfile to ”/mydlls/animal.dll” for AIX.
         *
         */

   char *animalFile = ”/mydlls/animal.dll”;  /* AIX filename */

   somEnvironmentNew();
   animalClass = _somFindClsInFile (SOMClassMgrObject,
                                    somIdFromString(animalName),
                                    0, 0,
                                    animalFile);
   myAnimal = _somNew (animalClass);
   somPrintf(”The class of myAnimal is %s.\n”,
       _somGetClassName(myAnimal));
   _somFree(myAnimal);
}
/*
Output from this program:
The class of myAnimal is Animal.
*/
```

Original Class

SOMClassMgr

Related Information

Methods: somFindClass

somGetInitFunction Method

Purpose

Obtains the name of the function that initializes the SOM classes in a class library.

IDL Syntax

string somGetInitFunction ( );

Note: For backward compatibility, this method does *not* take an **Environment** parameter.

Description

The **somGetInitFunction** method supplies the name of the initialization function for OS/2 class libraries (DLLs) that contain more than one SOM class. The default implementation returns the value of the global variable **SOMClassInitFuncName**, which by default is set to the value “SOMInitModule”.

For AIX, the name of the class initialization function is not important, since AIX class libraries should always be constructed as shared libraries with a designated entry point which can be executed automatically by the loader when the class is loaded. Consequently, the result of this method is not significant on AIX.

Similarly, if an OS/2 class library (DLL) has been constructed with a DLL initialization function assigned by the linker, you can choose to invoke the <*className*>**NewClass** functions for all of the classes in the DLL during DLL initialization. In this case (as on AIX), there is no need to export a “SOMInitModule” function. On the other hand, if your compiler does not provide a convenient mechanism for creating a DLL initialization function, you can elect to export a function named “SOMInitModule” (or whatever name is ultimately returned by the **somGetInitFunction** method).

The OS/2 **SOMClassMgrObject**, after loading a class library, will invoke the method **somGetInitFunction** to obtain the name of a possible initialization function. If this name has been exported by the class library just loaded, the **SOMClassMgrObject** calls this function to initialize the classes in the library. If the name has not been exported by the DLL, the **SOMClassMgrObject** then looks for an exported name of the form <*className*>**NewClass**, where <*className*> is the name of the class supplied with the method that caused the DLL to be loaded. If the DLL exports this name, it is invoked to create the named class.

On Windows, the SOM class manager does *not* call **SOMInitModule**. It must be called from the default Windows DLL initialization function, LibMain. This call is made indirectly through the **SOM_ClassLibrary** macro.

Regardless of the technique employed, the **SOMClassMgrObject** expects that all classes packaged in a single class library will be created during this sequence.

This method is generally not invoked directly by users. User-defined subclasses of **SOMClassMgr**, however, can override this method.

Parameters

receiver	Usually **SOMClassMgrObject** (or a pointer to an instance of a user-supplied subclass of **SOMClassMgr**).

Return Value

The **somGetInitFunction** method returns a string that names the initialization function of class libraries. By default, this name is the value of the global variable **SOMClassInitFuncName**, the default value of which is **SOMInitModule**.

Original Class

SOMClassMgr

Related Information

Methods: somFindClass, **somFindClsInFile**

Functions: SOMInitModule

Macros: SOM_ClassLibrary

somGetRelatedClasses Method

Purpose

Returns an array of class objects that were all registered during the dynamic loading of a class.

IDL Syntax

SOMClass * somGetRelatedClasses (in SOMClass *classObj***);**

Note: For backward compatibility, this method does *not* take an **Environment** parameter.

Description

The **somGetRelatedClasses** method returns an array of class objects that were all registered during the dynamic loading of the specified class. These classes are considered to define an affinity group. Any class is a member of at most one affinity group. The affinity group returned by this call is the one containing the class identified by the *classObj* parameter.

The first element in the array is either the class that caused the group to be loaded, or the special value –1, which means that the class manager is currently in the process of unregistering and deleting the affinity group (only class-manager objects would ever see this value). The remainder of the array consists of pointers to class objects, ordered in reverse chronological sequence to that in which they were originally registered. This list includes the given argument, *classObj*, as one of its elements, as well as the class that caused the group to be loaded (also given by the first element of the array). The array is terminated by a NULL pointer as the last element.

Use **SOMFree** to release the array when it is no longer needed. If the supplied class was not dynamically loaded, it is not a member of any affinity group and NULL is returned.

Parameters

receiver	Usually a pointer to **SOMClassMgrObject**, or a pointer to an instance of a user-defined subclass of **SOMClassMgr**.
classObj	A pointer to a **SOMClass** object.

Return Value

The **somGetRelatedClasses** method returns a pointer to an array of pointers to class objects, or NULL, if the specified class was not dynamically loaded.

Example

```
#include <som.h>
SOMClass myClass, *relatedClasses;
string className;
long i;

className = SOMClass_somGetName (myClass));
relatedClasses = SOMClassMgr_somGetRelatedClasses
                                    (SOMClassMgrObject, myClass);
if (relatedClasses && *relatedClasses) {
    somPrintf (”Class=%s, related classes are: ”, className);
    for (i=1; relatedClasses[i]; i++)
        somPrintf (”%s ”,SOMClass_somGetName
(relatedClasses[i]));
    somPrintf (”\n”);
    somPrintf (”Class that caused loading was %s\n”,
        relatedClasses[0] == (SOMClass) -1 ? ”-1” :
            SOMClass_somGetName (relatedClasses[0]));
    SOMFree (relatedClasses);
} else
    somPrintf (”No classes related to %s\n”, className);
```

Original Class

SOMClassMgr

Related Information

Methods: somGetInitFunction

somLoadClassFile Method

Purpose

Dynamically loads a class.

IDL Syntax

SOMClass somLoadClassFile (
in somId *classId*,
in long *majorVersion*,
in long *minorVersion*,
in string *file***);**

Note: For backward compatibility, this method does *not* take an **Environment** parameter.

Description

The **SOMClassMgr** object uses the **somLoadClassFile** method to load a class dynamically during the execution of **somFindClass** or **somFindClsInFile**. A SOM class object representing the class is expected to be created and registered as a result of this method.

The **somLoadClassFile** method can be overridden to load or create classes dynamically using your own mechanisms. If you simply wish to change the name of the procedure that is called to initialize the classes in a library, override **somGetInitFunction** instead.

This method is generally not invoked directly by users. Instead, use **somFindClass** or **somFindClsInFile**.

Parameters

receiver	Usually **SOMClassMgrObject** (or a pointer to an instance of a user-supplied subclass of **SOMClassMgr**).
classId	The **somId** representing the name of the class to load.
majorVersion	The major version number used to check the compatibility of the class's implementation with the caller's expectations.
minorVersion	The minor version number used to check the compatibility of the class's implementation with the caller's expectations.
file	The name of the dynamically linked library file containing the class. The name can be either a simple, unqualified name (without any extension) or a fully qualified (or path) file name, as appropriate for your operating system. For example, on OS/2, *file* could be `c:\myhome\myapp\basename.dll` or else `basename` (but not `basename.dll`).

Return Value

The **somLoadClassFile** method returns a pointer to the class object, or NULL if the class could not be loaded or the class object could not be created.

Original Class

SOMClassMgr

Related Information

Methods: somFindClass, **somFindClsInFile**, **somGetInitFunction**, **somUnloadClassFile**

somLocateClassFile Method

Purpose

Determines the file that holds a class to be dynamically loaded.

IDL Syntax

```
string  somLocateClassFile (
                             in somId classId,
                             in long majorVersion,
                             in long minorVersion);
```

Note: For backward compatibility, this method does *not* take an **Environment** parameter.

Description

The **SOMClassMgr** object uses the **somLocateClassFile** method when executing **somFindClass** to obtain the name of a file to use when dynamically loading a class. The default implementation consults the Interface Repository for the value of the *dllname* modifier of the class; if no *dllname* modifier was specified, the method simply returns the class name as the expected filename.

If you override the **somLocateClassFile** method in a user-supplied subclass of **SOMClassMgr,** the name you return can be either a simple, unqualified name without any extension or a fully qualified file name. Generally speaking, you would not invoke this method directly. It is provided to permit customization of subclasses of **SOMClassMgr** through overriding.

Parameters

receiver	Usually **SOMClassMgrObject** (or a pointer to an instance of a user-supplied subclass of **SOMClassMgr**).
classId	The **somId** representing the name of the class to locate.
majorVersion	The major version number used to check the compatibility of the class's implementation with the caller's expectations.
minorVersion	The minor version number used to check the compatibility of the class's implementation with the caller's expectations.

Return Value

The **somLocateClassFile** method returns the name of the file containing the class.

Original Class

SOMClassMgr

Related Information

Methods: somFindClass, **somFindClsInFile**, **somGetInitFunction**, **somLoadClassFile**, **somUnloadClassFile**

somMergeInto Method

Purpose

Transfers SOM class registry information to another **SOMClassMgr** instance.

IDL Syntax

void somMergeInto (in SOMClassMgr *target***);**

Note: For backward compatibility, this method does *not* take an **Environment** parameter.

Description

The **somMergeInto** method transfers the **SOMClassMgr** registry information from one object to another. The target object is required to be an instance of **SOMClassMgr** or one of its subclasses. At the completion of this operation, the target object can function as a replacement for the receiver. The receiver object (which is then in a newly uninitialized state) is placed in a mode where all methods invoked on it will be delegated to the target object. If the receiving object is the instance pointed to by the global variable **SOMClassMgrObject**, then **SOMClassMgrObject** is reassigned to point to the target object.

Subclasses of **SOMClassMgr** that override the **somMergeInto** method should transfer their section of the class manager object from the target to the receiver, then invoke their parent's **somMergeInto** method as the final step.

Invoke this method only if you are creating your own subclass of **SOMClassMgr**. Invoke **somMergeInto** from your override of the **SOMClassMgr**'s **somNew** method.

Parameters

receiver	Usually **SOMClassMgrObject** (or a pointer to an instance of a user-supplied subclass of **SOMClassMgr**).
target	A pointer to another instance of **SOMClassMgr** or one of its subclasses.

C Example

```
/*
 * The following example is a hypothetical
 * implementation of an override of the somNew method
 * in a subclass of SOMClassMgr.  It illustrates the
 * proper use of the somMergeInto method.
 */
SOM_Scope SOMAny * SOMLINK somNew (MySOMClassMgr somSelf)
{
    SOMAny *newInstance;
    static int firstTime = 1;
    /*
     * Permit only one instance of MySOMClassMgr to be created.
     */
    if (!firstTime)
        return (SOMClassMgrObject);
    newInstance = parent_SOMClassMgr_somNew (somSelf);
    /*
     * The next line will transfer the class registry
     * information from SOMClassMgrObject into our
     * new instance.
     */
    _somMergeInto (SOMClassMgrObject, newInstance);
    /* As a result of the above operation
     * SOMClassMgrObject is now set to point to the
     * new instance of MySOMClassMgr.
     */
    firstTime = 0;
    return (newInstance);
}
```

Original Class

SOMClassMgr

somRegisterClass Method

Purpose

Adds a class object to the SOM run-time class registry.

IDL Syntax

void somRegisterClass (in SOMClass *classObj***);**

Note: For backward compatibility, this method does *not* take an **Environment** parameter.

Description

The **somRegisterClass** method adds a class object to the SOM run-time class registry maintained by **SOMClassMgrObject**.

All SOM run-time class objects should be registered with the **SOMClassMgrObject**. This is done automatically during the execution of the **somClassReady** method as class objects are created.

Parameters

receiver	Usually **SOMClassMgrObject** (or a pointer to an instance of a user-supplied subclass of **SOMClassMgr**).
classObj	A pointer to the class object to add to the SOM class registry.

Original Class

SOMClassMgr

Related Information

Methods: somUnregisterClass

somSubstituteClass Method

Purpose

Causes the **somFindClass**, **somFindClsInFile**, and **somClassFromId** methods to substitute one class for another.

IDL Syntax

```
long  somSubstituteClass (
                              in string origClassName,
                              in string newClassName);
```

Note: For backward compatibility, this method does *not* take an **Environment** parameter.

Description

The **somSubstituteClass** method causes the **somFindClass**, **somFindClsInFile**, and **somClassFromId** methods to return the class named *newClassName* whenever they would normally return the class named *origClassName*. This effectively results in class *newClassName* replacing or substituting for class *origClassName*. For example, the <*origClassName*>**New** macro will subsequently create instances of *newClassName.*

Some restrictions are enforced to ensure that this works well. Both class *origClassName* and class *newClassName* must have been already registered before issuing this method, and *newClassName* must be an immediate child of *origClassName*. In addition (although not enforced), no instances should exist of either class at the time this method is invoked.

A convenience macro (**SOM_SubstituteClass**) is provided for C or C++ users. In one operation, it creates both the old and the new class and then substitutes the new one in place of the old. The use of both the **somSubstituteClass** method and the **SOM_SubstituteClass** macro is illustrated in the following example.

Parameters

receiver	Usually **SOMClassMgrObject** or a pointer to an instance of a user-defined subclass of **SOMClassMgr**.
origClassName	A NULL terminated string containing the old class name.
newClassName	A NULL terminated string containing the new class name.

Return Value

The **somSubstituteClass** method returns a value of zero to indicate success; a non-zero value indicates an error was detected.

C Example

```
#include ”student.h”
#include ”mystud.h”

/* Macro form */
SOM_SubstituteClass (Student, MyStudent);

/* Direct use of the method, equivalent to
 * the macro form above.
 */
{
SOMClass origClass, replacementClass;

origClass = StudentNewClass (Student_MajorVersion,
                                        Student_MinorVersion);
replacementClass = MyStudentNewClass (MyStudent_MajorVersion,
                                      MyStudent_MinorVersion);
SOMClassMgr_somSubstituteClass (
    SOMClass_somGetName (origClass),
    SOMClass_somGetName (replacementClass));
}
```

Original Class

SOMClassMgr

Related Information

Methods: somClassFromId, **somFindClass**, **somFindClsInFile**, **somMergeInto**

somUnloadClassFile Method

Purpose

Unloads a dynamically loaded class and frees the class's object.

IDL Syntax

long somUnloadClassFile (in SOMClass *class***);**

Note: For backward compatibility, this method does *not* take an **Environment** parameter.

Description

The **somUnregisterClass** method uses the **somUnloadClassFile** method to unload a dynamically loaded class. This releases the class's code and unregisters all classes in the same affinity group. (Use **somGetRelatedClasses** to find out which other classes are in the same affinity group.)

The class object is freed whether or not the class' s shared library could be unloaded. If the class was not registered, an error condition is raised and **SOMError** is invoked. This method is provided to permit user-created subclasses of **SOMClassMgr** to handle the unloading of classes by overriding this method. Do not invoke this method directly; instead, invoke **somUnregisterClass**.

Parameters

receiver	Usually **SOMClassMgrObject** (or a pointer to an instance of a user-supplied subclass of **SOMClassMgr**).
class	A pointer to the class to be unloaded.

Return Value

The **somUnloadClassFile** method returns 0 if the class was successfully unloaded; otherwise, it returns a system-specific non-zero error code from either the OS/2 **DosFreeModule** or the AIX **unload** system call or the Windows **FreeLibrary** system call.

Original Class

SOMClassMgr

Related Information

Methods: somClassFromId, **somRegisterClass**, **somUnregisterClass**, **somGetRelatedClasses**

somUnregisterClass Method

Purpose

Removes a class object from the SOM run-time class registry.

IDL Syntax

long somUnregisterClass (in SOMClass *class***);**

Note: For backward compatibility, this method does *not* take an **Environment** parameter.

Description

The **somUnregisterClass** method unregisters a SOM class and frees the class object. If the class was dynamically loaded, it is also unloaded using **somUnloadClassFile** (which causes its entire affinity group to be unloaded as well).

Parameters

receiver	Usually **SOMClassMgrObject** (or a pointer to an instance of a user-supplied subclass of **SOMClassMgr**).
class	A pointer to the class to be unregistered.

Return Value

The **somUnregisterClass** method returns 0 for a successful completion, or non-zero to denote failure.

Example

```
#include <som.h>

void main ()
{
    long rc;  /* Return code */
    SOMClass animalClass;

    /* The next 2 lines declare a static form of somId */
    string animalClassName = ”Animal”;
    somId animalId = &animalClassName;

    somEnvironmentNew ();
    animalClass = SOMClassMgr_somFindClass (SOMClassMgrObject,
                                             animalId, 0, 0);
    if (!animalClass) {
        somPrintf (”Could not load class.\n”);
        return;
    }
    rc = SOMClassMgr_somUnregisterClass (SOMClassMgrObject,
                                                animalClass);
    if (rc)
        somPrintf (”Could not unregister class, error code:
%ld.\n”,

rc);
    else
        somPrintf (”Class successfully unloaded.\n”);
}
```

Original Class

SOMClassMgr

Related Information

Methods: somLoadClassFile, **somRegisterClass**, **somUnloadClassFile**

SOMObject Class

SOMObject is the root class for all SOM classes. That is, all SOM classes must be subclasses of **SOMObject** or of some other class derived from **SOMObject**. **SOMObject** introduces no instance data, so objects whose classes inherit from **SOMObject** incur no size increase. They do inherit a suite of methods that provide the behavior required of all SOM objects. Three of these methods are typically overridden by any subclass that has instance data — **somDefaultInit**, **somDestruct**, and **somDumpSelfInt**. See the descriptions of these methods for more information.

File Stem

somobj

Base

None

Metaclass

SOMClass

Ancestor Classes

None

New Methods

Group: Initialization/Termination

somFree

somDefaultInit

somDestruct

somInit

somUninit

Group: Access

somGetClass

somGetClassName

somGetSize

Group: Testing

somIsA

somIsInstanceOf

somRespondsTo

Group: Dynamic

somDispatchA
somDispatchD
somDispatchL
somDispatchV
somDispatch
somClassDispatch
somCastObj
somResetObj

Group: Development Support

somDumpSelf

somDumpSelfInt

somPrintSelf

Overridden Methods

None

somCastObj Method

Purpose

Changes the behavior of an object to that defined by any ancestor of the true class of the object.

IDL Syntax

boolean somCastObj (in SOMClass *ancestor***);**

Description

The **somCastObj** method changes the behavior of an object so that its behavior will be that of an instance of the indicated ancestor class (with respect to any method supported by the ancestor). The behavior of the object on methods not supported by the ancestor remains unchanged.

This operation actually changes the class of the object (since an object's behavior is defined by its class). The name of the new class is derived from the initial name of the object's class and the name of the ancestor class, as illustrated in the following example.

The **somCastObj** method may be used on an object multiple times, always with the restriction that the ancestor class whose behavior is selected is actually an ancestor of the true (original) class of the object.

Parameters

receiver	A pointer to an object of type **SOMObject**.
ancestor	A pointer to a class that is an ancestor of the actual class of the *receiver*.

Return Value

The **somCastObj** method returns 1 (TRUE) if the operation is successful and 0 (FALSE) otherwise. The operation fails if *ancestor* is not actually an ancestor of the class of the object.

Example

```
#include <som.h>
main()
{
   SOMClassMgr cm = somEnvironmentNew();
   SOM_Test(1 == _somCastObj(cm, _SOMObject));
   _somDumpSelf(cm, 0));
   SOM_Test(1 == _somResetObj(cm));
   _somDumpSelf(cm, 0);
}

/* output:
 *  {An instance of class SOMClassMgr->SOMObject
 *   at address 20061268
 *  }
 *  {An instance of class SOMClassMgr at address 20061268
 *   ... <SOMClassMgr State Information> ...
 *  }
 */
```

Original Class

SOMObject

Related Information

Methods: somResetObj

somDefaultInit Method

Purpose

Initializes instance variables and attributes in a newly created object. Replaces **somInit** as the preferred method for default object initialization. For performance reasons, it is recommended that **somDefaultInit** always be overridden by classes.

Syntax

void somDefaultInit (inout somInitCtrl *ctrl* **);**

Description

Every SOM class is expected to support a set of initializer methods. This set will always include **somDefaultInit**, whether or not the class explicitly overrides **somDefaultInit**. All other initializer methods for a class must be explicitly introduced by the class. See Section 5.5, "Initializing and Uninitializing Objects," of the *SOMobjects Developer Toolkit Users Guide* for complete information on introducing new initializers.

The purpose of an initializer method supported by a class is first to invoke initializer methods of ancestor classes (those ancestors that are the class's **directinitclasses**) and then to place the instance variables and attributes introduced by the class into some consistent state by loading them with appropriate values. The result is that, when an object is initialized, each class that contributes to its implementation will run some initializer method. The **somDefaultInit** method may or may not be among the initializers used to initialize a given object, but it is always available for this purpose.

Thus, the **somDefaultInit** method may be invoked on a newly created object to initialize its instance variables and attributes. The **somDefaultInit** method is more efficient than **somInit** (the method it replaces), and it also prevents multiple initializer calls to ancestor classes. The **somInit** method is now considered obsolete when writing new code, although **somInit** is still supported.

To override **somDefaultInit**, the **implementation** section of the class's .idl file should include **somDefaultInit** with the **override** and **init** modifiers specified. (The **init** modifier signifies that the method is an *initializer* method.) No additional coding is required for the resulting **somDefaultInit** stub procedure in the implementation template file, unless the class implementor wishes to customize object initialization in some way.

If the .idl file does *not* explicitly override **somDefaultInit**, then by default a generic method procedure for **somDefaultInit** will be provided by the SOMobjects Toolkit. If invoked, this generic method procedure first invokes **somDefaultInit** on the appropriate ancestor classes, and then (for consistency with earlier versions of SOMobjects) calls any **somInit** code that may have been provided by the class (if **somInit** was overridden). Because the generic procedure for **somDefaultInit** is less efficient than the stub procedure that is provided when **somDefaultInit** is overridden, it is recommended that the .idl file always override **somDefaultInit.**

Note: It is *not* appropriate to override both **somDefaultInit** and **somInit**. If this is done, the **somInit** code will not be executed. The best way to convert an old class that overrides **somInit** to use of the more efficient **somDefaultInit** (if this is desired) is as follows: (1) Replace the **somInit** override in the class's .idl file with an override for **somDefaultInit**, (2) run the implementation template emitter to produce a stub procedure for **somDefaultInit**, and then (3) simply call the class's **somInit** procedure directly (*not* using a method invocation) from the **somDefaultInit** method procedure.

As mentioned previously, the object-initialization framework supported by SOMobjects allows a class to support additional initializer methods besides **somDefaultInit.** These additional initializers will typically include special-purpose arguments, so that objects of the class can be initialized with special capabilities or characteristics. For each new initializer method, the **implementation** section must include the method name with the **init** modifier. Also, the **directinitclasses** modifier can be used if, for some reason, the class implementor wants to control the order in which ancestor initializers are executed.

Notes: It is recommended that the method name for an *initializer* method include the class name as a prefix. A newly defined initializer method will include an implicit Environment argument if the class does not use a **callstyle=oidl** modifier.

Important: There are important constraints associated with modification of the procedure stubs for initializers. These are documented in Section 5.5 of the *SOMobjects Developer Toolkit Users Guide*.

Parameters

receiver	A pointer to an object**.**
ctrl	A pointer to a **somInitCtrl** data structure. SOMobjects uses this data structure to control the initialization of the ancestor classes, thereby ensuring that no ancestor class receives multiple initialization calls.

Example

```
// SOM IDL
#include <Animal.idl>

interface Dog : Animal
{
    implementation {
        releaseorder: ;
            somDefaultInit: override, init;
        };
};
```

Original Class

SOMObject

Related Information

Methods: somDestruct

somDestruct Method

Purpose

Uninitializes the receiving object, and (if so directed) frees object storage after uninitialization has been completed. Replaces **somUninit** as the preferred method for uninitializing objects. For performance reasons, it is recommended that **somDestruct** always be overridden. Not normally invoked directly by object clients.

Syntax

void somDestruct (in octet *dofree*, **inout somDestructCtrl** *ctrl***);**

Description

Every class must support the **somDestruct** method. This is accomplished either by overriding **somDestruct** (in which case a specialized stub procedure will be generated in the implementation template file), or else SOMobjects will automatically provide a generic procedure that implements **somDestruct** for the class. The generic procedure calls **somUninit** (if this was overridden) to perform local uninitialization, then completes execution of the method appropriately.

Because the specialized stub procedure generated by the template emitter is more efficient than the generic procedure provided when **somDestruct** is not overridden, it is recommended that **somDestruct** always be overridden. The stub procedure that is generated in this case requires no modification for correct operation. The only modification appropriate within this stub procedure is to uninitialize locally introduced instance variables. See Section 5.5, “Initializing and Uninitializing Objects,” of the *SOMobjects Developer Toolkit Users Guide* for further details.

Uninitialization with **somDestruct** executes as follows: For any given class in the ancestor chain, **somDestruct** first uninitializes that class’s introduced instance variables (if this is appropriate), and then calls the next ancestor class’s implementation of **somDestruct**, passing 0 (that is, false) as the interim *dofree* argument. Then, after all ancestors of the given class have been uninitialized, if the class’s own **somDestruct** method were originally invoked with *dofree* as 1 (that is, true), then that object’s storage is released.

Note: It is *not* appropriate to override both **somDestruct** and **somUninit**. If this is done, the **somUninit** code will not be executed. The best way to convert an old class that overrides **somUninit** to use of the more efficient **somDestruct** (if this is desired) is as follows: (1) Replace the **somUninit** override in the class’s .idl file with an override for **somDestruct**, (2) run the emitter to produce a stub procedure for **somDestruct** in the implementation template file, and then (3) simply call the class’s **somUninit** procedure directly (*not* using a method invocation) from the **somDestruct** procedure.

Parameters

receiver	A pointer to an object.
dofree	A boolean indicating whether the caller wants the object storage freed after uninitialization of the current class has been completed. Passing 1 (true) indicates the object storage should be freed.
ctrl	A pointer to a **somDestructCtrl** data structure. SOMobjects uses this data structure to control the uninitialization of the ancestor classes, thereby ensuring that no ancestor class receives multiple uninitialization calls. If a user invokes **somDestruct** on an object directly, a NULL (that is, zero) ctrl pointer can be passed. This instructs the receiving code to obtain a **somDestructCtrl** data structure from the class of the object.

Example

```
// SOM IDL
#include <Animal.idl>

interface Dog : Animal
{
    implementation {
        releaseorder: ;
            somDestruct: override;
        };
};
```

Original Class

SOMObject

Related Information

Methods: somDefaultInit

somDispatch, somClassDispatch Methods

Purpose

Invokes a method using dispatch method resolution. The **somDispatch** method is designed to be overridden. The **somClassDispatch** method is not generally overridden.

IDL Syntax

boolean somDispatch (
  out somToken *retValue*,
  in somId *methodId*,
  in va_list *args***);**

boolean somClassDispatch (
  in SOMClass *clsObj*,
  out somToken *retValue*,
  in somId *methodId*,
  in va_list *args***);**

Note: For backward compatibility, these methods do *not* take an **Environment** parameter.

Description

Both **somDispatch** and **somClassDispatch** perform method resolution to select a method procedure, and then invoke this procedure on *args*. The “somSelf” argument for the selected method procedure (called the “target object,” in the following text, to distinguish it from the receiver of the **somDispatch** or **somClassDispatch** method call) is the first argument included in the va_list, *args*.

For **somDispatch**, method resolution is performed using the class of the receiver; for **somClassDispatch**, method resolution is performed using the argument class, *clsObj*. Because **somClassDispatch** uses *clsObj* for method resolution, a programmer invoking **somDispatch** or **somClassDispatch** should assure that the class of the target object is either derived from or is identical to the class used for method resolution; otherwise, a run-time error will likely result when the target object is passed to the resolved procedure. Although not necessary, the receiver is usually also the target object.

The **somDispatch** and **somClassDispatch** methods supersede the **somDispatch***X* methods. Unlike the **somDispatch***X* methods, which are restricted to few return types, the **somDispatch** and **somClassDispatch** methods make no assumptions concerning the result returned by the method to be invoked. Thus, **somDispatch** and **somClassDispatch** can be used to invoke methods that return structures. The **somDispatch***X* methods now invoke **somDispatch**, so overriding **somDispatch** serves to override the **somDispatch***X* methods as well.

Parameters

receiver	A pointer to the object whose class will be used for method resolution by **somDispatch**.
clsObj	A pointer to the class that will be used for method resolution by **somClassDispatch**.
retValue	The address of the area in memory where the result of the invoked method procedure is to be stored. The caller is responsible for allocating enough memory to hold the result of the specified method. When dispatching methods that return no result (that is, void), a NULL may be passed as this argument.
methodId	A **somId** identifying the method to be invoked. A string representing the method name can be converted to a **somId** using the **somIdFromString** function.
args	A **va_list** containing the arguments to be passed to the method identified by *methodId*. The arguments must include a pointer to the target object as the first entry. As a convenience for C and C++ programmers, SOM's language bindings provide a varargs invocation macro for va_list methods (such as **somDispatch** and **somClassDispatch).** The following example illustrates this.

Return Value

A boolean representing whether or not the method was successfully dispatched is returned. The reason for this is that **somDispatch** and **somClassDispatch** use the function **somApply** to invoke the resolved method procedure, and **somApply** requires an apply stub for successful execution. In support of old class binaries SOM does not consider a NULL apply stub to be an error. As a result, somApply may fail. If this happens, then false is returned; otherwise, true is returned.

C Example

Given class *Key* that has an attribute *keyval* of type **long** and an overridden method for **somPrintSelf** that prints the value of the attribute (as well as the information printed by **SOMObject**'s implementation of **somPrintSelf**), the following client code invokes methods on *Key* objects using **somDispatch** and **somClassDispatch**. (The *Key* class was defined with the **callstyle=oidl** class modifier, so the **Environment** argument is not required of its methods.)

```
#include <key.h>

main()
{
  SOMObject obj;
  long k1 = 7, k2;
  Key myKey = KeyNew();
  va_list push, args = SOMMalloc(8);
  somId setId = somIdFromString(”_set_keyval”);
  somId getId = somIdFromString(”_get_keyval”);
  somId prtId = somIdFromString(”_somPrintSelf”);

  /* va_list invocation of setkey and getkey : */
  push = args;
  va_arg(push, SOMObject) = myKey;
  va_arg(push, long) = k1;
  SOMObject_somDispatch(myKey,(somToken*)0,setId,args);
  push = args;
  va_arg(push, SOMObject) = myKey;
  SOMObject_somDispatch(myKey,(somToken*)&k2,getId,args);
  printf(”va_list _set_keyval and _get_keyval: %i\n”, k2);

  /* varargs invocation of setkey and getkey : */
  _somDispatch(myKey, (somToken*)0, setId, myKey, k1);
  _somDispatch(myKey, (somToken*)&k2, getId, myKey);
  printf(”varargs _set_keyval and _get_keyval: %i\n”, k2);

  /* illustrate somclassDispatch ”casting” (use varargs form) */
  printf(”somPrintSelf on myKey as a Key:\n”);
  _somClassDispatch(myKey,_Key,(somToken*)&obj2,prtId,myKey,0);

  printf(”somPrintSelf on myKey as a SOMObject:\n”);

_somClassDispatch(myKey,_SOMObject,(somToken*)&obj,prtId,myKey,0)
;
  SOMFree(args); SOMFree(setId); SOMFree(getId); SOMFree(prtId);
 _somFree(myKey);
}
```

This program produces the following output:

```
va_list _set_keyval and _get_keyval: 7
varargs _set_keyval and _get_keyval: 7
somPrintSelf on myKey as a Key:
{An instance of class Key at address 2005B2F8}
    -- with key value 7
somPrintSelf on myKey as a SOMObject:
{An instance of class Key at address 2005B2F8}
```

Original Class

SOMObject

Related Information

Functions: somApply

somDispatch*X* Methods (Obsolete)

Purpose

Invoke a method using dispatch method resolution. These methods are obsolete.

IDL Syntax

```
somToken somDispatchA (
                        in somId methodId,
                        in somId descriptor,
                        in va_list args);

double  somDispatchD (
                        in somId methodId,
                        in somId descriptor,
                        in va_list args);

long  somDispatchL (
                        in somId methodId,
                        in somId descriptor,
                        in va_list args);

void  somDispatchV (
                        in somId methodId,
                        in somId descriptor,
                        in va_list args);
```

Note: For backward compatibility, these methods do *not* take an **Environment** parameter.

Description

The **somDispatch*X*** methods are superseded by the more general **somDispatch** method, and are retained solely for backward compatibility.

The **somDispatch*X*** methods invoke on the receiving object the method identified by *methodId*, with arguments specified by *args*. The target object for the method invocation is the receiving object, which is *not* included in the arguments.

Parameters

receiver	A pointer to the object on which the dispatched method is invoked.
methodId	A **somId** that represents the method to be invoked.
descriptor	A **somId** that represents the types of the arguments being passed in the *args* **va_list**. This parameter is not used in the current implementation, so a NULL value can be substituted.
args	A **va_list** containing the arguments to be passed to the method identified by *methodId*. The arguments do not include the target for the dispatched method.

Return Value

Four families of return values are supported, corresponding to the four forms of the **somDispatch***X* method. The **somDispatch***X* method chosen should have a return type compatible with the result of the method identified by *methodId*. Within each of the four families, only the largest representation is supported.

The four families are:

Pointer	**somDispatchA** returns an address as a **somToken**.
Floating point	**somDispatchD** returns a floating point number as a **double.**
Integer	**somDispatchL** returns an integer as a **long**.
Void	**somDispatchV** returns **void**. It is used for methods that do not return a result.

Original Class

SOMObject

Related Information

Methods: somDispatch

Functions: somApply

somDumpSelf Method

Purpose

Writes out a detailed description of the receiving object. Intended for use by object clients. Not generally overridden.

IDL Syntax

void somDumpSelf (in long *level***);**

Note: For backward compatibility, this method does *not* take an **Environment** parameter.

Description

The **somDumpSelf** method performs some initial setup, and then invokes the **somDumpSelfInt** method to write a detailed description of the receiver, including its state.

Parameters

receiver	A pointer to the object to be dumped.
level	The nesting level for describing compound objects. It must be greater than or equal to 0. All lines in the description will be preceded by “2 * level” spaces.

Example

See the **somDumpSelfInt** method.

Original Class

SOMObject

Related Information

Methods: somDumpSelfInt

somDumpSelfInt Method

Purpose

Outputs the internal state of an object. Intended to be overridden by class implementors. Not intended to be directly invoked by object clients.

IDL Syntax

void somDumpSelfInt (in long *level***);**

Note: For backward compatibility, this method does *not* take an **Environment** parameter.

Description

The **somDumpSelfInt** method should be overridden by a class implementor, to write out the instance data stored in an object. This method is invoked by the **somDumpSelf** method, which is used by object clients to output the state of an object.

The procedure used to override this method for a new class should begin by calling the parent class form of this method on each of the class parents, and should then write a description of the instance variables introduced by new class. This will result in a description of all the class's instance variables. The C and C++ implementation bindings provide a convenient macro for performing parent method calls on all parents, as illustrated in the following examples.

The character output routine pointed to by **SOMOutCharRoutine** should be used for output. The **somLPrintf** function is especially convenient for this, since level is handled appropriately.

Parameters

receiver	A pointer to the object to be dumped.
level	The nesting level for describing compound objects. It must be greater than or equal to 0. All lines in the description should be preceded by "2 * level" spaces.

C Example

Following is a method overriding **somDumpSelfInt** for class "List", which has two attributes, *val* (which is a **long**) and *next* (which is a pointer to a "List" object).

```
SOM_Scope void   SOMLINK somDumpSelfInt(List somSelf, int level)
{
    ListData *somThis = ListGetData(somSelf);
    Environment *ev = somGetGlobalEnvironment();

    List_parents_somDumpSelfInt(somSelf, level);
    somLPrintf(level, "This item: %i\n", __get_val(somSelf, ev);
    somLPrintf(level, "Next item: \n");
    if (__get_next(somSelf, ev) != (List) NULL)
        _somDumpSelfInt(__get_next(somSelf, ev), level+1);
    else
        somLPrintf(level+1, "NULL\n");
}
```

Following is a client program that invokes the **somDumpSelf** method on “List” objects:

```
#include <list.h>

main()
{
   List L1, L2;
   long x = 7, y = 13;
   Environment *ev = somGetGlobalEnvironment();

   L1 = ListNew();
   L2 = ListNew();
   __set_val(L1, ev, x);
   __set_next(L1, ev, (List) NULL);
   __set_val(L2, ev, y);
   __set_next(L2, ev, L1);

   _somDumpSelf(L2,0);

   _somFree(L1);
   _somFree(L2);
}
```

Following is the output produced by this program:

```
{An instance of class List at 0x2005EA8
 This item: 13
 Next item:
   1 This item: 7
   1 Next item:
     2 NULL
}
```

Original Class

SOMObject

Related Information

Methods: somDumpSelf, **somPrintSelf**

somFree Method

Purpose

Releases the storage used by an object and frees the object. Intended for use by object clients. Not generally overridden.

IDL Syntax

void somFree ( );

Note: For backward compatibility, this method does *not* take an **Environment** parameter.

Description

The **somFree** method releases the storage containing the receiver object by calling the method **somDeallocate**. No future references should be made to the receiver once this is done. Before releasing storage, **somFree** calls **somUninit** to allow storage pointed to the object to be freed.

The **somFree** method should not be called on objects created by **somRenew**, thus the method is normally only used by code that also created the object.

Note: SOM also supplies a function, **SOMFree**, which is used to free a block of memory. This function should not be used on objects.

Parameters

receiver	A pointer to the object to be freed.

C Example

```
#include <animal.h>

void main()
{
   Animal myAnimal;
   /*
    * Create an object.
    */
   myAnimal = AnimalNew();

   /* ... */

   /* Free it when finished. */
   _somFree(myAnimal);
}
```

Original Class

SOMObject

Related Information

Methods: somNew, somNewNoInit, **somUninit**

Functions: SOMFree

somGetClass Method

Purpose

Returns a pointer to an object's class object. Not generally overridden.

IDL Syntax

SOMClass somGetClass ( );

Note: For backward compatibility, this method does *not* take an **Environment** parameter.

Description

somGetClass obtains a pointer to the receiver's class object. The **somGetClass** method is typically not overridden.

Important : For C and C++ programmers, SOM provides a **SOM_GetClass** macro that performs the same function. This macro should only be used **only** when absolutely necessary (that is, when a method call on the object is not possible), since it bypasses whatever semantics may be intended for the somGetClass method by the implementor of the receiver's class. Even class implementors do not know whether a special semantics for this method is inherited from ancestor classes. If you are unsure of whether the method or the macro is appropriate, you should use the method call.

Parameters

receiver	A pointer to the object whose class is desired.

Return Value

A pointer to the object's class object.

C Example

```
#include <animal.h>
main()
{
  Animal myAnimal;
  int numMethods;
  SOMClass animalClass;

  myAnimal = AnimalNew ();
  animalClass = _somGetClass (myAnimal);
  SOM_Test(animalClass == _Animal);
}
```

Original Class

SOMObject

Related Information

Macros: SOM_GetClass

somGetClassName Method

Purpose

Returns the name of the class of an object. Not generally overridden.

IDL Syntax

string somGetClassName ( );

Note: For backward compatibility, this method does *not* take an **Environment** parameter.

Description

The **somGetClassName** method returns a pointer to a zero-terminated string that gives the name of the class of an object.

This method is not generally overridden; it simply invokes **somGetName** on the class of the receiver. Refer to **somGetName** for more information on the returned string,

Parameters

receiver	A pointer to the object whose class name is desired.

Return Value

The **somGetClassName** method returns a pointer to the name of the class.

C Example

```
#include <animal.h>
main()
{
  Animal myAnimal;
  SOMClass animalClass;
  char *className;

  myAnimal = AnimalNew();
  className = _somGetClassName(myAnimal);
  somPrintf("Class name: %s\n", className);
  _somFree(myAnimal);
}
/*
Output from this program:
Class name: Animal
*/
```

Original Class

SOMObject

Related Information

Methods: somGetName

somGetSize Method

Purpose

Returns the size of an object. Not generally overridden.

IDL Syntax

long somGetSize ( );

Note: For backward compatibility, this method does *not* take an **Environment** parameter.

Description

The **somGetSize** method returns the total amount of contiguous space used by the receiving object.

The value returned reflects only the amount of storage needed to hold the SOM representation of the object. The object might actually be using or managing additional space outside of this area.

The **somGetSize** method is not generally overridden.

Parameters

receiver	A pointer to the object whose size is desired.

Return Value

The **somGetSize** method returns the size, in bytes, of the receiver.

C Example

```
#include <animal.h>
void main()
{
  Animal myAnimal;
  int animalSize;
  myAnimal = AnimalNew();
  animalSize = _somGetSize(myAnimal);
  somPrintf(”Size of animal (in bytes): %d\n”, animalSize);
  _somFree(myAnimal);
}
/*
Output from this program:
Size of animal (in bytes): 8
*/
```

Original Class

SOMObject

Related Information

Methods: somGetInstancePartSize, **somGetInstanceSize**

somInit Method

Purpose

Initializes instance variables or attributes in a newly created object. Designed to be overridden.

Note: The newer **somDefaultInit** method is suggested instead.

IDL Syntax

void somInit ( );

Note: For backward compatibility, this method does *not* take an **Environment** parameter.

Description

The **somInit** method is invoked to cause a newly created object to initialize its instance variables or attributes.

Note: The newer **somDefaultInit** method performs object initialization more efficiently and is now the preferred approach for overriding initialization in an implementation file. (The **somInit** method still executes correctly as before.)

Because instances of **SOMObject** do not have any instance data, the default implementation does nothing. It is provided as a convenience to class implementors so that initialization of objects can be done in a uniform way across all classes (by overriding **somInit**). This method is called automatically by **somNew** during object creation.

A companion method, **somUninit**, is called whenever an object is freed. These two methods should be designed to work together, with **somInit** priming an object for its first use, and **somUninit** preparing the object for subsequent release.

If objects of your class contain instance variables or attributes, override the **somInit** method to initialize the instance variables or attributes when instances of the class are created. When overriding this method, always call all parent (base) classes' versions of this method *before* doing your own initialization, as follows:

1. The overriding implementation should invoke the parent method for *each* parent. For users of the C or C++ implementation bindings, this can be done in either of two ways:
   a. By calling a <*className*>**_parents_**<*methodName*> macro (which automatically invokes all parent methods) or
   b. By calling the <*className*>**_parent_**<*parentName*>_<*methodName*> macro on each parent separately.

      For more information on parent method calls, see the topic "Extending the Implementation Template" in Chapter 5, "Implementing Classes in SOM," of the *SOM Toolkit User's Guide.*
2. The code must be written so that it can be executed multiple times without harm on the same object. This is necessary because, under multiple inheritance, parent method calls that progress up the inheritance hierarchy may encounter the same ancestor class more than once (where different inheritance paths "join" when followed backward). A check can be made to determine whether a particular invocation of **somInit** is the first on a given object by examining the contents of its instance variables; all the instance variables of a newly created SOM object are set to zero before **somInit** is invoked on that object.

More information and examples on object initialization (especially regarding the **somDefaultInit** method) are given in the topic “Initializing and Uninitializing Objects” in Chapter 5, “Implementing Classes in SOM,” of the *SOM Toolkit User’s Guide.*

Parameters

receiver	A pointer to the object to be initialized.

C Example

Following is the implementation for a class *Animal* that introduces an attribute *sound* of type *string* and overrides **somInit** and **somUninit**, along with a main program that creates and then frees an instance of class *Animal*:

```
#define Animal_Class_Source
 #include <animal.ih>
 #include <string.h>

 SOM_Scope void SOMLINK somInit (Animal somSelf)
  {
      AnimalData *somThis = AnimalGetData (somSelf);
      Environment *ev = somGetGlobalEnvironment();
      Animal_parents_somInit (somSelf);
      if (!__get_sound(somSelf, ev)) {
         __set_sound(somSelf, ev, SOMMalloc(100));
         strcpy (__get_sound(somSelf, ev), ”Unknown Noise”);
         somPrintf (”New Animal Initialized\n”);
      }
  }

 SOM_Scope void SOMLINK somUninit (Animal somSelf)
  {
      AnimalData *somThis = AnimalGetData (somSelf);
      Environment *ev = somGetGlobalEnvironment();
      if (__get_sound(somSelf, ev)) {
        SOMFree(__get_sound(somSelf, ev);
        __set_sound(somSelf, ev, (char*)0);
        somPrintf (”Animal Uninitialized\n”);
        Animal_parents_somUninit (somSelf);
      }
  }

/* main program */
   #include <animal.h>
   void main()
   {
      Animal myAnimal;
      myAnimal = AnimalNew ();
      _somFree (myAnimal);
   }

/*
Program output:
New Animal Initialized
Animal Uninitialized
*/
```

Original Class

SOMObject

Related Information

Methods: somDefaultInit, **somNew**, **somRenew**, **somDestruct**, **somUninit**

somIsA Method

Purpose

Tests whether an object is an instance of a given class or of one of its subclasses. Not generally overridden.

IDL Syntax

boolean somIsA (in SOMClass *aClass***);**

Note: For backward compatibility, this method does *not* take an **Environment** parameter.

Description

Use the **somIsA** method to determine if an object can be treated like an instance of *aClass*. SOM guarantees that if **somIsA** returns true, then the *receiver* will respond to all (static or dynamic) methods supported by *aClass*.

Parameters

receiver	A pointer to the object to be tested.
aClass	A pointer to the class that the object should be tested against.

Return Value

The **somIsA** methods returns 1 (true) if the receiving object is an instance of the specified class or (unlike **somIsInstanceOf**) of any of its descendant classes, and 0 (false) otherwise.

C Example

```
#include <dog.h>
/* --------------------------------
   : Dog is derived from Animal.
   -------------------------------- */
main()
{
  Animal myAnimal;
  Dog myDog;
  SOMClass animalClass;
  SOMClass dogClass;

  myAnimal = AnimalNew();
  myDog = DogNew();
  animalClass = _somGetClass (myAnimal);
  dogClass = _somGetClass (myDog);
  if (_somIsA (myDog, animalClass))
     somPrintf (”myDog IS an Animal\n”);
  else
     somPrintf (”myDog IS NOT an Animal\n”);
  if (_somIsA (myAnimal, dogClass))
     somPrintf (”myAnimal IS a Dog\n”);
  else
     somPrintf (”myAnimal IS NOT a Dog\n”);
  _somFree (myAnimal);
  _somFree (myDog);
}
/*
Output from this program:
myDog IS an Animal
myAnimal IS NOT a Dog
*/
```

Original Class

SOMObject

Related Information

Methods: somDescendedFrom, **somIsInstanceOf**, **somRespondsTo**, **somSupportsMethod**

somIsInstanceOf Method

Purpose

Determines whether an object is an instance of a specific class. Not generally overridden.

IDL Syntax

boolean somIsInstanceOf (in SOMClass *aClass***);**

Note: For backward compatibility, this method does *not* take an **Environment** parameter.

Description

Use the **somIsInstanceOf** method to determine if an object is an instance of a specific class. This method tests an object for inclusion in one specific class. It is equivalent to the expression:

```
(aClass == somGetClass (receiver))
```

Parameters

receiver	A pointer to the object to be tested.
aClass	A pointer to the class that the object should be an instance of.

Return Value

The **somIsInstanceOf** method returns 1 (true) if the receiving object is an instance of the specified class, and 0 (false) otherwise.

C Example

```
#include <dog.h>
/* --------------------------------
   : Dog is derived from Animal.
   -------------------------------- */
main()
{
  Animal myAnimal;
  Dog myDog;
  SOMClass animalClass;
  SOMClass dogClass;

  myAnimal = AnimalNew ();
  myDog = DogNew ();
  animalClass = _somGetClass (myAnimal);
  dogClass = _somGetClass (myDog);
  if (_somIsInstanceOf (myDog, animalClass))
     somPrintf (”myDog is an instance of Animal\n”);
  if (_somIsInstanceOf (myDog, dogClass))
     somPrintf (”myDog is an instance of Dog\n”);
  if (_somIsInstanceOf (myAnimal, animalClass))
     somPrintf (”myAnimal is an instance of Animal\n”);
  if (_somIsInstanceOf (myAnimal, dogClass))
     somPrintf (”myAnimal is an instance of Dog\n”);
  _somFree (myAnimal);
  _somFree (myDog);
}
/*
Output from this program:
myDog is an instance of Dog
myAnimal is an instance of Animal
*/
```

Original Class

SOMObject

Related Information

Methods: somDescendedFrom, **somIsA**

somPrintSelf Method

Purpose

Outputs a brief description that identifies the receiving object. Designed to be overridden.

IDL Syntax

SOMObject somPrintSelf ( );

Note: For backward compatibility, this method does *not* take an **Environment** parameter.

Description

somPrintSelf should output a brief string containing key information useful to identify the receiver object, rather than a complete dump of the receiver object state as provided by **somDumpSelfInt**. The **somPrintSelf** method should use the character output routine **SOMOutCharRoutine** (or any of the **somPrintf** functions) for this purpose. The default implementation outputs the name of the receiver object's class and the receiver's address in memory.

Because the most specific identifying information for an object will often be found within instance data introduced by the class of an object, it is likely that a class implementor that overrides this method will not need to invoke parent methods in order to provide a useful string identifying the receiver object.

Parameters

receiver	A pointer to the object to be described.

Return Value

The **somPrintSelf** method returns a pointer to the receiver object as its result.

C Example

```
#include <animal.h>
main()
{
  Animal myAnimal;
  myAnimal = AnimalNew ();
  /* ... */
  _somPrintSelf (myAnimal);
  _somFree (myAnimal);
}
/*
Output from this program:

{An instance of class Animal at address 0001CEC0}
*/
```

Original Class

SOMObject

Related Information

Methods: somDumpSelf, **somDumpSelfInt**

somResetObj Method

Purpose

Resets an object’s class to its true class after use of the **somCastObj** method.

Syntax

boolean somResetObj ( );

Description

The **somResetObj** method resets an object’s class to its true class after use of the **somCastObj** method.

Parameters

receiver	A pointer to a SOM object.

Return Value

The **somResetObj** method returns 1 (TRUE) always.

Example

```
#include <som.h>
main()
{
   SOMClassMgr cm = somEnvironmentNew();
   SOM_Test(1 == _somCastObj(cm, _SOMObject));
   _somDumpSelf(cm, 0));
   SOM_Test(1 == _somResetObj(cm));
   _somDumpSelf(cm, 0);
}

/* output:
 *  {An instance of class SOMClassMgr->SOMObject
 *   at address 20061268
 *  }
 *  {An instance of class SOMClassMgr at address 20061268
 *   ... <SOMClassMgr State Information> ...
 *  }
 */
```

Original Class

SOMObject

Related Information

Methods: somCastObj

somRespondsTo Method

Purpose

Tests whether the receiving object supports a given method. Not generally overridden.

IDL Syntax

boolean somRespondsTo (in somId *methodId***);**

Note: For backward compatibility, this method does *not* take an **Environment** parameter.

Description

The **somRespondsTo** method tests whether a specific (static or dynamic) method can be invoked on the receiver object. This test is equivalent to determining whether the class of the receiver *supports* the specified method on its instances.

Parameters

receiver	A pointer to the object to be tested.
methodId	A **somId** that represents the name of the desired method.

Return Value

The **somRespondsTo** method returns TRUE if the specified method can be invoked on the receiving object, and FALSE otherwise.

C Example

```
/* ----------------------------------------------
   : Animal supports a setSound method;
         Animal does not support a doTrick method.
   ---------------------------------------------- */
#include <animal.h>
main()
{
  Animal myAnimal;
  char *methodName1 = ”setSound”;
  char *methodName2 = ”doTrick”;

  myAnimal = AnimalNew();
  if (_somRespondsTo(myAnimal, SOM_IdFromString(methodName1)))
     somPrintf(”myAnimal responds to %s\n”, methodName1);
  if (_somRespondsTo(myAnimal, SOM_IdFromString(methodName2)))
     somPrintf(”myAnimal responds to %s\n”, methodName2);
  _somFree(myAnimal);
}
/*
Output from this program:
myAnimal responds to setSound
*/
```

Original Class

SOMObject

Related Information

Methods: somSupportsMethod

somUninit Method

Purpose

Un-initializes the receiving object. Designed to be overridden by class implementors. Not normally invoked directly by object clients.

IDL Syntax

void somUninit ( );

Note: For backward compatibility, this method does *not* take an **Environment** parameter.

Description

The **somUninit** method performs the inverse of object initialization. Class implementors that introduce instance data that points to allocated storage should override somUninit so allocated storage can be freed when an object is freed.

This method is called automatically by **somFree** to clean up anything necessary (such as extra storage dynamically allocated to the object) before **somFree** releases the storage allocated to the object itself.

Code responsible for freeing an object must first know that there will be no further references to this object. Once this is known, this code would normally invoke **somFree** (which calls **somUninit**). In cases where **somRenew** was used to create an object instance, however, **somFree** cannot be called (for example, the storage containing the object may simply be a location on the stack), and in this case, **somUninit** must be called explicitly.

When overriding this method, always call the parent-class versions of this method *after* doing your own un-initialization. Furthermore, just as with **somInit**, because your method may be called multiple times (due to multiple inheritance), you should zero out references to memory that is freed, and check for zeros before freeing memory and calling the parent methods.

Parameters

receiver	A pointer to the object to be un-initialized.

C Example

Following is the implementation for a class *Animal* that introduces an attribute *sound* of type *string* and overrides **somInit** and **somUninit**, along with a main program that creates and then frees an instance of class *Animal*:

```
#define Animal_Class_Source
 #include <animal.ih>
 #include <string.h>

 SOM_Scope void SOMLINK somInit (Animal somSelf)
  {
      AnimalData *somThis = AnimalGetData (somSelf);
      Environment *ev = somGetGlobalEnvironment();
      Animal_parents_somInit (somSelf);
      if (!__get_sound(somSelf, ev)) {
         __set_sound(somSelf, ev, SOMMalloc(100));
         strcpy (__get_sound(somSelf, ev), "Unknown Noise");
         somPrintf ("New Animal Initialized\n");
      }
  }

 SOM_Scope void SOMLINK somUninit (Animal somSelf)
  {
      AnimalData *somThis = AnimalGetData (somSelf);
      Environment *ev = somGetGlobalEnvironment();
      if (__get_sound(somSelf, ev)) {
        SOMFree(__get_sound(somSelf, ev);
        __set_sound(somSelf, ev, (char*)0);
        somPrintf ("Animal Uninitialized\n");
        Animal_parents_somUninit (somSelf);
      }
 }

/* main program */
   #include <animal.h>
   void main()
   {
      Animal myAnimal;
      myAnimal = AnimalNew ();
      _somFree (myAnimal);
   }

/*
Program output:
New Animal Initialized
Animal Uninitialized
*/
```

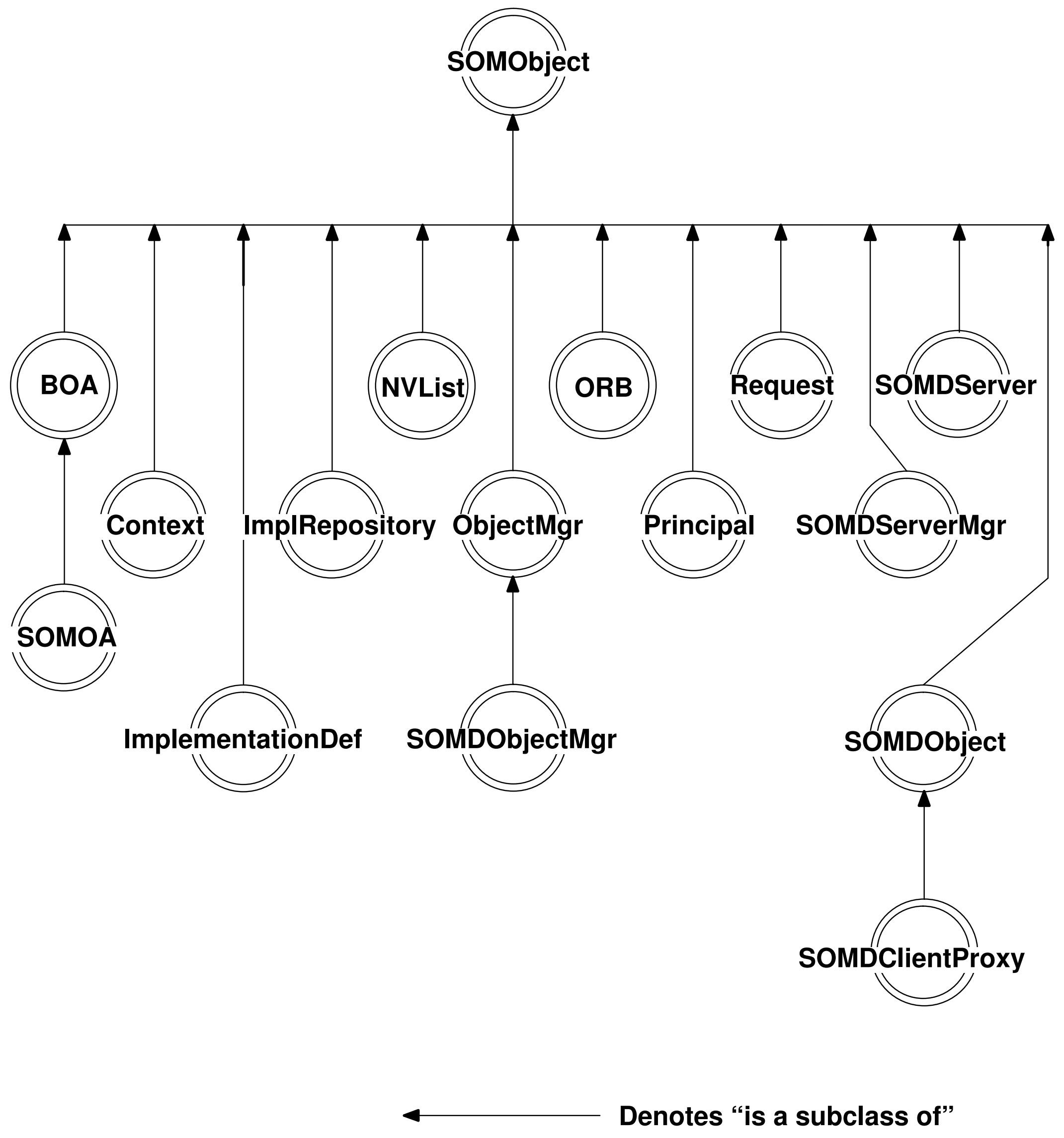
Original Class

SOMObject

Related Information

Methods: somInit, **somNew**, **somRenew**

Chapter 2. DSOM Framework Reference



DSOM Framework Class Organization

Notes

The following information should be considered when using the Distributed SOM (DSOM) framework.

DSOM and CORBA

Distributed SOM (DSOM) is a framework that supports access to objects in a distributed application. DSOM can be viewed as both:

- An extension to basic SOM facilities
- An implementation of the “Object Request Broker” (ORB) technology defined by the Object Management Group (OMG), in the Common Object Request Broker Architecture (CORBA) specification and standard, Revision 1.1. The CORBA 1.1 specification is published by x/Open and the Object Management Group (OMG).

One of the primary contributions of CORBA is the specification of basic runtime interfaces for writing portable, distributable object-oriented applications. SOM and DSOM implement those runtime interfaces, according to the CORBA specification.

In addition to the published CORBA 1.1 interfaces, it was necessary for DSOM to introduce several of its own interfaces, in those areas where:

- CORBA 1.1 did not specify the full interface (for example, **ImplementationDef**, **Principal**),
- CORBA 1.1 did not address the function specified by the interface (for example, “lifecycle” services for object creation and deletion), or
- The functionality of a CORBA 1.1 interface has been enhanced by DSOM.

Any such interfaces have been noted on the reference page for each DSOM *class*.

A Note on Method Naming Conventions

The SOM Toolkit frameworks (including DSOM) and CORBA have slightly different conventions for naming methods. Methods introduced by the SOM Toolkit frameworks use prefixes to indicate the framework to which each method belongs, and use capitalization to separate words in the method names (for example, **somdFindServer**). Methods introduced by CORBA have no prefixes, are all lower case, and use underscores to separate words in the method names (such as, **impl_is_ready**).

DSOM, more than the other SOM Toolkit frameworks, uses a mix of both conventions. The method and class names introduced by CORBA 1.1 are implemented as specified, for application portability. Methods introduced by DSOM to enhance a CORBA-defined class also use the CORBA naming style. The SOM Toolkit convention for method naming is used for non-CORBA classes which are introduced by DSOM.

get_next_response Function

Purpose

Returns the next **Request** object to complete, after starting multiple requests in parallel.

C Syntax

ORBStatus get_next_response (
Environment* *env*,
Flags *response_flags*,
Request **req* **);**

Description

The **get_next_response** function returns a pointer to the next **Request** object to complete after starting multiple requests in parallel. This is a synchronization function used in conjunction with the **send_multiple_requests** function. There is no specific order in which requests will complete.

If the *response_flags* field is set to 0, this function will not return until the next request completion. If the caller does not want to become blocked, the RESP_NO_WAIT flag should be specified.

Parameters

env	A pointer to the **Environment** structure for the caller.
response_flags	A **Flags** (unsigned long) variable, used to indicate whether the caller wants to wait for the next request to complete (0), or not wait (RESP_NO_WAIT).
req	A pointer to a **Request** object variable. The address of the next **Request** object which completes is returned in the **Request** variable.

Return Value

The **get_next_response** function may return a non-zero **ORBStatus** value, which indicates a DSOM error code. (See the *SOM Toolkit User's Guide* for more information on DSOM error codes.)

Example

See the example for the **send_multiple_requests** function.

Related Information

Functions: send_multiple_requests

Methods: send, **get_response**, **invoke**

This function is described in section 6.3, “Deferred Synchronous Routines”, of the CORBA 1.1 specification.

ORBfree Function

Purpose

Frees memory allocated by DSOM for return values and **out** arguments.

C Syntax

void ORBfree (void* *ptr***);**

Description

The **ORBfree** function is used to free memory for method return values or **out** arguments which are placed in memory allocated by DSOM (versus the calling program). For example, strings, arrays, sequence buffers, and “any” values are returned in memory which is dynamically allocated by DSOM.

Parameters

ptr	A pointer to memory that has been dynamically allocated by DSOM for a method return value or **out** argument.

Example

```
#include <somd.h>
#include <myobject.h> /* provided by user */

MyObject obj;
Environment ev;
string str;

/* assume myMethod has the following IDL declaration
 * in the MyObject interface:
 *
 *  void myMethod(out string s);
 */
_myMethod(obj, &ev, &str);
...

/* free storage */
ORBfree(str);
```

Related Information

Functions: SOMD_NoORBfree

This function is described in section 5.16, “Argument Passing Considerations”, and section 5.17, “Return Result Passing Considerations”, of the CORBA 1.1 specification.

send_multiple_requests Function

Purpose

Initiates multiple **Requests** in parallel.

C Syntax

ORBStatus send_multiple_requests (
Request *reqs***[ ],**
Environment* *env***,**
long *count***,**
Flags *invoke_flags* **);**

Description

The **send_multiple_requests** function initiates multiple **Requests** “in parallel”. (The actual degree of parallelism is system dependent.) Each **Request** object is created using the **create_request** method, defined on **SOMDClientProxy**. Like the **send** method, this function returns to the caller immediately without waiting for the **Requests** to finish. The caller waits for the request responses using the **get_next_response** function.

Parameters

reqs The address of an array of **Requests** objects which are to be initiated in parallel.

env A pointer to the **Environment** structure for the caller.

count The number of **Request** objects in *reqs.*

invoke_flags A **Flags** (unsigned long) value, used to indicate the following options:

INV_NO_RESPONSE
Indicates the caller does not intend to get any results or **out** parameter values from any of the requests. The requests can be treated as if they are **oneway** operations.

INV_TERM_ON_ERR
If one of the requests causes an error, the remaining requests are not sent.

The above flag values may be “or”-ed together.

Return Value

The **send_multiple_requests** function may return a non-zero **ORBStatus** value, which indicates a DSOM error code. (See the *SOM Toolkit User’s Guide* for more information on DOSM error codes.)

Example

```
#include <somd.h>

/* sum a set of values in parallel */
int parallel_sum(Environment *ev, int n, SOMDObject *objs)
{
 int index, sum = 0;
 Request *next;
 Request *reqs = (Request*) SOMMalloc(n * sizeof(Request));
 NamedValue *results = (NamedValue*)
         SOMMalloc(n * sizeof(Namedvalue));

 for (i=0; i < n; i++)
  (void) _create_request((Context *)NULL, "_get_count", NULL,
          &(result[i]), &(reqs[i]), (Flags)0);

 (void) send_multiple_requests(reqs, ev, n, (Flags)0);

 for (i=0, i < n; i++) {
  (void) get_next_response(ev, (Flags)0, &next);
  index = (next - reqs);
  sum += *((int*)results[index].argument._value);
 }

 return(sum);
}
```

Related Information

Functions: get_next_response

Methods: send, **get_response**, **invoke**

This function is described in section 6.3, “Deferred Synchronous Routines”, of the CORBA 1.1 specification.

somdExceptionFree Function

Purpose

Frees the memory held by the exception structure within an **Environment** structure, regardless of whether the exception was returned by a local or a remote method call.

C Syntax

void somdExceptionFree (Environment ***_ev_);**

Description

The **somdExceptionFree** function frees the memory held by the exception structure within an **Environment** structure, regardless of whether the exception was returned by a local or a remote method call.

When a DSOM client program invokes a remote method and the method returns an exception in the **Environment** structure, it is the client's responsibility to free the exception. This is done by calling either **exception_free** or **somdExceptionFree** on the **Environment** structure in which the exception was returned. (The two functions are equivalent. The **exception_free** function name is #defined in the **som.h** or **som.xh** file to provide strict CORBA compliance of function names.) There is a similar function, **somExceptionFree**, available for SOM programmers; DSOM programmers, however, can use **somdExceptionFree** to free all exceptions (regardless of whether they were returned from a local or a remote method call).

Parameters

_ev_	The **Environment** structure whose exception information is to be freed.

Example

```
X_foo(x, ev, 23);  /* make a remote method call */
if (ev->major != NO_EXCEPTION)
{
  printf(”foo exception = %s\n”, somExceptionId(ev));

  /* ... handle exception ... */

  somdExceptionFree(ev);  /* free exception */
}
```

Related Information

Functions: somExceptionFree, **somExceptionId**, **somExceptionValue**, **somSetException** (all SOM kernel functions)

Data structures: Environment (**somcorba.h**)

SOMD_Init Function

Purpose

Initializes DSOM in the calling process.

C Syntax

void SOMD_Init (Environment* *env***);**

Description

Initializes DSOM in the calling process. This function should be called before any other DSOM functions or methods. This function should only be invoked (a) at the beginning of a DSOM program (client or server), to initialize the program, or (b) after **SOMD_Uninit** has been invoked, to reinitialize the program. If the program has already been initialized with **SOMD_Init**, then invoking **SOMD_Init** again has no effect.

An effect of calling **SOMD_Init** is that the global variables **SOMD_ObjectMgr, SOMD_ImplRepObject,** and **SOMD_ORBObject**, are initialized with pointers to the (single) instances of the **SOMDObjectMgr, ImplRepository,** and **ORB** objects.

Parameters

env	A pointer to the **Environment** structure for the caller.

Return Value

None. (However, the global variables **SOMD_ObjectMgr, SOMD_ImplRepObject,** and **SOMD_ORBObject** are set implicitly.)

Example

```
#include <somd.h>

Environment ev;

/* initialize Environment */
SOM_InitEnvironment(&ev);

/* initialize DSOM runtime */
SOMD_Init(&ev);
...

/* Free DSOM resources */
SOMD_Uninit(&ev);
```

Related Information

SOM Toolkit User's Guide.

SOMD_NoORBfree Function

Purpose

Specifies to DSOM that the client program will use the **SOMFree** function to free memory allocated by DSOM, rather than using the **ORBfree** function.

C Syntax

void SOMD_NoORBfree ();

Description

The **SOMD_NoORBfree** function is used in a DSOM client program to specify to DSOM that the client program will use the **SOMFree** function to free memory allocated by DSOM, rather than using the **ORBfree** function.

Typically, a DSOM client program will use **SOMFree** to free memory returned from local method calls and **ORBfree** to free memory returned from remote method calls. The **SOMD_NoORBfree** function allows programmers to use a single function (**SOMFree**) to free blocks of memory, regardless of whether they were allocated locally or by DSOM in response to a remote method call.

SOMD_NoORBfree, if used, should be called just after calling **SOMD_Init** in the client program. In response to this call, DSOM will not keep track of the memory it allocates for the client. Instead, it will assume that the client program will be responsible for walking all data structures returned from remote method calls, calling **SOMFree** for each block of memory within.

Example

```
SOMD_Init();
SOMD_NoORBfree();

/* rest of client program */
```

Related Information

Functions: ORBfree, **SOMFree**

SOMD_RegisterCallback Function

Purpose

Registers a callback function for handling DSOM request events.

C Syntax

void SOMLINK SOMD_RegisterCallback (SOMEEMan *emanObj*, **EMRegProc** ***func*);**

Description

When writing event-driven applications where there are event sources other than DSOM requests (for example, user input, mouse clicks, and so forth), DSOM cannot be given exclusive control of the "main loop," such as when **execute_request_loop** is called. Instead, the application should use the Event Management (EMan) framework to register and process all application events.

The **SOMD_RegisterCallback** function is used to register a user-supplied DSOM event handler function with EMan. The caller need only supply an address of the event handler function, and the instance of the EMan object — the details of registration are implemented by **SOMD_RegisterCallback**.

Callback functions should have the SOMLINK keyword explicitly specified, except on Windows. Using an explicit SOMLINK keyword on Windows will preclude the ability of an application to support multiple instances.

Note: The function **SOMD_RegisterCallback** must be declared with "system linkage" on OS/2.

Parameters

emanObj A pointer to an instance of **SOMEEman**, the Event Manager object.

func A pointer to an event handler function which will be called by EMan whenever a DSOM request arrives. This function must have the following prototype (equivalent to the **EMRegProc** type defined in the **eman.h** file):

```
#ifdef __OS2__
 #pragma linkage(func, system)
#endif

void SOMLINK func (SOMEEvent event, void *eventData)
/* On Windows, using the SOMLINK keyword precludes
 * the support of multiple instances. */
```

Example

```
#include <somd.h>
#include <eman.h>

#ifdef __OS2__
 #pragma linkage(SOMD_RegisterCallback, system)
 #pragma linkage(DSOMEventCallBack, system)
#endif

/* On Windows, this example would omit the SOMLINK keyword. */

void SOMLINK DSOMEventCallBack (SOMEEvent event, void *eventData)
{
 Environment ev;
 SOM_InitEnvironment(&ev);

 _execute_request_loop (SOMD_SOMOAObject, &ev, SOMD_NO_WAIT);
}

main()
{
 ...
 eman = SOMEEmanNew();
 SOMD_RegisterCallback(eman, DSOMEventCallBack);

 _someProcessEvents(eman, &ev); /* main loop */
 ...
}
```

Related Information

See Chapter 12 of the *SOM Toolkit User's Guide* for a description of the Event Management (EMan) framework, for writing event-driven applications.

SOMD_Uninit Function

Purpose

Free system resources allocated for use by DSOM.

C Syntax

void SOMD_Uninit (Environment* *env***);**

Description

Frees system resources (such as, shared memory segments, semaphores) allocated to the calling process for use by DSOM. This function should be called before a process exits, to ensure system resources are reused.

No DSOM functions or methods should be called after **SOMD_Uninit** has been called. After **SOMD_Uninit** is called, the program can be reinitialized by calling **SOMD_Init**. (**SOMD_Uninit** would then need to be called again before program termination, to uninitialize the program.)

Parameters

env	A pointer to the **Environment** structure for the caller.

Example

```
#include <somd.h>

Environment ev;

/* initialize Environment */
SOM_InitEnvironment(&ev);

/* initialize DSOM runtime */
SOMD_Init(&ev);
...
/* Free DSOM resources */
SOMD_Uninit(&ev);
```

Related Information

See Chapter 6 on DSOM in the *SOM Toolkit User's Guide.*

Context_delete Macro

Purpose

Deletes a **Context** object.

Syntax

ORBStatus Context_delete (
Context *ctxobj*,
Environment **env*,
Flags *del_flag*);

Description

The **Context_delete** macro deletes the specified **Context** object. This macro maps to the **destroy** method of the **Context** class.

Parameters

ctxobj	A pointer to the **Context** object to be deleted.
env	A pointer to the **Environment** structure for the caller.
del_flag	A bitmask (unsigned long). If the flag CTX_DELETE_DESCENDANTS is specified, the macro deletes the specified **Context** object and all of its descendant **Context** objects. A zero value indicates that the flag is not set.

Expansion

Context_destroy (*ctxobj, env, del_flag* **)**

Example

```
#include <somd.h>

Environment ev;
Context cxt, newcxt;
long rc;
...
/* get the process’ default Context */
rc = _get_default_context(SOMD_ORBObject, &ev, &cxt);

/* make newcxt a child Context of the default Context (cxt) */
rc = _create_child(cxt, &ev, ”myContext”, &newcxt);
...
/* assuming no descendent Contexts have been
 * created from newcxt, we can destroy newcxt with flags=0
 */
rc = Context_delete(newcxt, &ev, (Flags) 0);
```

Related Information

Methods: Context_destroy

Request_delete Macro

Purpose

Deletes the memory allocated by the ORB for a **Request** object.

Syntax

ORBStatus Request_delete (
                                        Request *reqobj*,
                                        Environment **env*);

Description

The **Request_delete** macro deletes the specified **Request** object and all associated memory. This macro maps to the **destroy** method of the **Request** class.

Parameters

reqobj	A pointer to the **Request** object to be deleted.
env	A pointer to the **Environment** structure for the caller.

Expansion

Request_destroy (*reqobj, env* **)**

Example

```
#include <somd.h>
#include <repostry.h>
#include <intfacdf.h>
#include <foo.h> /* provided by user */

/* assume following method declaration in interface Foo:
 *      long methodLong  (in long inLong,inout long inoutLong);
 * then the following code sends a request to execute the call:
 *      result = methodLong(fooObj, &ev, 100,200);
 * using the DII without waiting for the result. Then, later,
 * waits for and then uses the result.
 */
Environment ev;
NVList arglist;
long rc;
Foo fooObj;
Request reqObj;
NamedValue result;

/* see the Example code for invoke to see how the request
 * is built
 */

/* Create the Request, reqObj */
rc = _create_request(fooObj, &ev, (Context *)NULL, ”methodLong”,
       arglist, &result, &reqObj, (Flags)0);

/* Finally, send the request */
rc = _send(reqObj, &ev, (Flags)0);

/* do some work, i.e. don’t wait for the result */

/* wait here for the result of the request */
rc = _get_response(reqObj, &ev, (Flags)0);

/* use the result */
if (result->argument._value == 9600) {...}

/* throw away the reqObj */
Request_delete(reqObj, &ev);
```

Related Information

Methods: Request_destroy

BOA Class

Description

The Basic Object Adapter (**BOA**) defines the basic interfaces that a server process uses to access services of an Object Request Broker like DSOM. The **BOA** defines methods for creating and exporting object references, registering implementations, activating implementations and authenticating requests.

For more information on the Basic Object Adapter, refer to Chapter 9 in the CORBA 1.1 specification.

Note: DSOM treats the **BOA** interface as an *abstract* class, which merely defines basic runtime interfaces (introduced in the CORBA specification) but does not implement those interfaces. Thus, there is no point in instantiating a **BOA** object. If a **BOA** object is created, any methods invoked on it will return a NO_IMPLEMENT exception. Instead, the SOM Object Adapter (**SOMOA**) subclass provides DSOM implementations for **BOA** methods. When a **BOA** method is invoked on the **SOMOA** object, the desired behavior will occur.

File Stem

boa

Base

SOMObject

Metaclass

SOMMSingleInstance

Ancestor Classes

SOMObject

Subclasses

SOMOA

New Methods

change_implementation

create

deactivate_impl

deactivate_obj

dispose

get_id

get_principal

impl_is_ready

obj_is_ready

set_exception

change_implementation Method

Purpose

Changes the implementation associated with the referenced object. *(Not implemented.)*

IDL Syntax

```
void  change_implementation (
                              in SOMDObject obj,
                              in ImplementationDef impl);
```

Description

The **change_implementation** method is defined by the CORBA specification, *but has a null implementation in DSOM.* This method always returns a NO_IMPLEMENT exception.

In CORBA 1.1, the **change_implementation** method is provided to allow an application to change the implementation definition of an object.

However, in DSOM, the **ImplementationDef** identifies the server which implements an object. In these terms, changing an object's implementation (that is, server) would result in a change in the object's location. In DSOM, moving objects from one server to another is considered an application-specific task, and hence, no default implementation is provided.

It *is* possible, however, to change the program which implements an object's server, or change the class library which implements an object's class. To modify the program associated with an **ImplementationDef**, use the **update_impldef** method defined on **ImplRepository.** To change the implementation of an object's class, replace the corresponding class library with a new (upward-compatible) one.

Parameters

receiver	A pointer to a **BOA** (**SOMOA**) object for the server.
env	A pointer to the **Environment** structure for the method caller.
obj	A pointer to the **SOMDObject** object which refers to the application object whose implementation is to be changed.
impl	A pointer to the **ImplementationDef** object representing the new implementation of the application object.

Return Value

The **SOMOA** implementation always returns a NO_IMPLEMENT exception, with a minor code of SOMDERROR_NotImplemented.

Original Class

BOA

create Method

Purpose

Creates a “reference” for a local application object which can be exported to remote clients.

IDL Syntax

```
typedef sequence<octet,1024> ReferenceData;    // in somdtype.idl

SOMDObject  create (
                    in ReferenceData id,
                    in InterfaceDef intf,
                    in ImplementationDef impl);
```

Description

The **create** method creates a **SOMDObject** which is used as a “reference” to a local application object. An object reference is simply an object which is used to refer to another target object — one may think of it as an “ID”, “link”, or “handle.” Object references are important in DSOM in that their values can be externalized (that is, can be represented in a string form) for transmission between processes, storage in files, and so on. In DSOM, the proxy objects in client processes are remote object references.

To create an object reference, the caller specifies the **ImplementationDef** of the calling process, the **InterfaceDef** of the target application object, and up to 1024 bytes of **ReferenceData** which is used by the application to identify and activate the application object. When subsequent method calls specify the object reference as a parameter, the application will use the reference to find and/or activate the referenced object.

Note that (as specified in CORBA 1.1) each call to **create** returns a unique object reference, even if the same parameters are used in subsequent calls. For each reference, the **ReferenceData** is stored in the reference data file (and backup file, if any) for the server.

The **SOMOA** class introduces a **change_id** method which allows a server to modify the **ReferenceData** of one of its references. (The **change_id** method is *not* in the CORBA 1.1 specification.)

Ownership of the returned **SOMDObject** is transferred to the caller.

Parameters

receiver	A pointer to a **BOA** (**SOMOA**) object for the server.
env	A pointer to the **Environment** structure for the method caller.
id	A pointer to the **ReferenceData** structure containing application-specific information describing the target object.
intf	A pointer to the **InterfaceDef** object which describes the interface of the target object.
impl	A pointer to the **ImplementationDef** object which describes the application (server) process which implements the target object.

Return Value

The **create** method returns a pointer to a **SOMDObject** which refers to a local application object.

Example

```
#include <somd.h>
#include <repostry.h>
#include <intfacdf.h>

Environment ev;
ReferenceData id;
InterfaceDef intfdef;
SOMDObject objref;
string fname; /* a file name to be saved with reference */
...
/* create the id for the reference */
id._maximum = id._length = strlen(fname)+1;
id._buffer = (string) SOMMalloc(strlen(fname)+1);
strcpy(id._buffer,fname);

/* get the interface def object for interface Foo*/
intfdef = _lookup_id(SOM_InterfaceRepository, &ev, ”Foo”);

objref = _create(SOMD_SOMOAObject,
     &ev, id, intfdef, SOMD_ImplDefObject);
...
```

Original Class

BOA

Related Information

Methods: change_id, **create_constant**, **create_SOM_ref**, **dispose**, **get_id**

deactivate_impl Method

Purpose

Indicates that a server implementation is no longer ready to process requests.

IDL Syntax

void deactivate_impl (
in ImplementationDef *impl***);**

Description

The **deactivate_impl** method indicates that the implementation is no longer ready to process requests.

Parameters

receiver	A pointer to a **BOA** (**SOMOA**) object for the server.
env	A pointer to the **Environment** structure for the method caller.
impl	A pointer to the **ImplementationDef** object representing the implementation to be deactivated.

Example

```
#include <somd.h>

ORBStatus rc;

/* server initialization code ... */

/* signal DSOM that server is ready */
_impl_is_ready(SOMD_SOMOAObject, &ev, SOMD_ImplDefObject);

for(rc = 0;rc==0;) {
  rc = _execute_next_request(SOMD_SOMOAObject, &ev, waitFlag);
  /* perform app specific code between messages here, e.g.,*/
  numMessagesProcessed++;
}

/* signal DSOM that server is deactivated */
_deactivate_impl(SOMD_SOMOAObject, &ev, SOMD_ImplDefObject);
```

Original Class

BOA

Related Information

Methods: impl_is_ready, **activate_impl_failed**, **execute_next_request**, **execute_request_loop**

deactivate_obj Method

Purpose

Indicates that an object server is no longer ready to process requests. *(Not implemented.)*

IDL Syntax

void deactivate_obj (
in SOMDObject *obj***);**

Description

The **deactivate_obj** method is defined by the CORBA specification, *but has a null implementation in DSOM.* This method always returns a NO_IMPLEMENT exception.

CORBA 1.1 distinguishes between servers that implement many objects ("shared"), versus servers that implement a single object ("unshared"). The **deactivate_obj** method is meant to be used by unshared servers, to indicate that the object (that is, server) is no longer ready to process requests.

DSOM does not distinguish between servers that implement a single object versus servers that implement multiple objects, so this method has no implementation.

Parameters

receiver	A pointer to a **BOA** (**SOMOA**) object for the server.
env	A pointer to the **Environment** structure for the method caller.
obj	A pointer to a **SOMDObject** which identifies the object (server) to be deactivated.

Original Class

BOA

Related Information

Methods: deactivate_impl, **impl_is_ready**, **obj_is_ready**

dispose Method

Purpose

Destroys an object reference.

IDL Syntax

void dispose (
in SOMDObject *obj***);**

Description

The **dispose** method disposes of an object reference.

Parameters

receiver	A pointer to a **BOA** object for the server.
env	A pointer to the **Environment** structure for the method caller.
obj	A pointer to the object reference to be destroyed.

Example

```
#include <somd.h>
#include <repostry.h>
#include <intfacdf.h>

SOMDObject objref;
ReferenceData id;
InterfaceDef intfdef;
...
objref =
 _create(SOMD_SOMOAObject, &ev, id, intfdef, SOMD_ImplDefObject);
...
_dispose(SOMD_SOMOAObject, &ev, objref);
```

Original Class

BOA

Related Information

Methods: create, **create_constant**, **create_SOM_ref**, **get_id**

get_id Method

Purpose

Returns reference data associated with the referenced object.

IDL Syntax

ReferenceData get_id (
in SOMDObject *obj***);**

Description

The **get_id** method returns the reference data associated with the referenced object.

Parameters

receiver	A pointer to a **BOA** (**SOMOA**) object for the server.
env	A pointer to the **Environment** structure for the method caller.
obj	A pointer to a **SOMDObject** object for which to return the **ReferenceData**.

Return Value

The **get_id** method returns a **ReferenceData** structure associated with the referenced object.

Example

```
#include <somd.h>
#include <repostry.h>
#include <intfacdf.h>

SOMDObject objref;
ReferenceData id1, id2;
InterfaceDef intfdef;
...
objref =
 _create(SOMD_SOMOAObject,&ev, id1, intfdef, SOMD_ImplDefObject);
...
/* get the ReferenceData from a SOMDObject */
id2 = _get_id(SOMD_SOMOAObject, &ev, objref);
```

Original Class

BOA

Related Information

Methods: create, **create_constant**, **dispose**

get_principal Method

Purpose

Returns the ID of the principal that issued the request.

IDL Syntax

Principal get_principal (
in SOMDObject *obj*,
in Environment* *req_ev***);**

Description

The **get_principal** method returns the ID of the principal that issued a request.

Parameters

receiver	A pointer to a **BOA** (**SOMOA**) object for the server.
env	A pointer to the **Environment** structure for the method caller.
obj	A pointer to the object reference which is the target of the method call.
req_ev	A pointer to the **Environment** object passed as input to the request.

Return Value

The **get_principal** method returns a pointer to a **Principal** object which identifies the user and host from which a request originated.

Example

```
#include <somd.h>

/* assumed context: inside a method implementation */
void methodBody(SOMObject *somSelf, Environment *ev, ...)
{
 Principal p;
 SOMDObject selfRef;
 Environment localev;

 SOMInitEnvironment(&localev);

 /* get a reference to myself from the server object */
 selfRef =
   somdRefFromSOMObj(SOMD_ServerObject, &ev, somSelf);

 /* get principal information from the SOMOA */
 p = _get_principal(SOMD_SOMOAObject, &localev, selfRef, ev);

 printf(”user = %s, host = %s\n”,
   __get_userName(p), __get_hostName(p));
 ...
}
```

Original Class

BOA

Related Information

Classes: Principal

impl_is_ready Method

Purpose

Indicates that the implementation is ready to process requests.

IDL Syntax

void impl_is_ready (
    in ImplementationDef *impl***);**

Description

The **impl_is_ready** method Indicates that the implementation is ready to process requests.

Parameters

receiver	A pointer to a **BOA** (**SOMOA**) object for the server.
env	A pointer to the **Environment** structure for the method caller.
impl	A pointer to the **ImplementationDef** object indicating which implementation is ready.

Example

```
#include <somd.h> /* needed by all servers */

main(int argc, char **argv)
{
 Environment ev;
 SOM_InitEnvironment(&ev);

 /* Initialize the DSOM run-time environment */
 SOMD_Init(&ev);

 /* Retrieve its ImplementationDef from the Implementation
  Repository by passing its implementation ID as a key */
 SOMD_ImplDefObject =
  _find_impldef(SOMD_ImplRepObject, &ev, argv[1]);

 /* Tell DSOM that the server is ready to process requests */
 _impl_is_ready(SOMD_SOMOAObject, &ev, SOMD_ImplDefObject);
 ...
}
```

Original Class

BOA

Related Information

Methods: deactivate_impl, **activate_impl_failed**, **obj_is_ready**, **execute_request_loop**, **execute_next_request**

obj_is_ready Method

Purpose

Indicates that an object (server) is ready to process requests. *(Not implemented.)*

IDL Syntax

void obj_is_ready (
in SOMDObject *obj*,
in ImplementationDef *impl*);

Description

The **obj_is_ready** method is defined by the CORBA specification, *but has a null implementation in DSOM.* This method always returns a NO_IMPLEMENT exception.

CORBA 1.1 distinguishes between servers that implement many objects ("shared"), versus servers that implement a single object ("unshared"). The **obj_is_ready** method is meant to be used by unshared servers, to indicate that the object (that is, server) is ready to process requests.

DSOM does not distinguish between servers that implement a single object versus servers that implement multiple objects, so this method has no implementation.

Parameters

receiver	A pointer to a **BOA** (**SOMOA**) object for the server.
env	A pointer to the **Environment** structure for the method caller.
obj	A pointer to a **SOMDObject** which identifies the object (server) that is ready.
impl	A pointer to the **ImplementationDef** object representing the object that is ready.

Original Class

BOA

Related Information

Methods: impl_is_ready, **deactivate_impl**, **deactivate_obj**, **activate_impl_failed**

set_exception Method

Purpose

Returns an exception to a client.

IDL Syntax

void set_exception (
in exception_type *major*,
in string *except_name*,
in void* *param***);**

Description

The **set_exception** method returns an exception to the client. The *major* parameter can have one of three possible values:

NO_EXCEPTION — indicates a normal outcome of the operation. It is not necessary to invoke **set_exception** to indicate a normal outcome; it is the default behavior if the method simply returns.

USER_EXCEPTION — indicates a user-defined exception.

SYSTEM_EXCEPTION — indicates a system-defined exception.

Parameters

receiver	A pointer to a **BOA** (**SOMOA**) object for the server.
env	A pointer to the **Environment** structure for the method caller.
major	One of the exception types NO_EXCEPTION**,** USER_EXCEPTION**,** or SYSTEM_EXCEPTION.
except_name	A **string** representing the exception type identifier.
param	A pointer to the associated data.

Example

```
#include <somd.h>
#include <myobject.h> /* provided by user */

/* assuming following IDL declarations in the MyObject interface:
 *  exception foo;
 *  void myMethod() raises (BadCall);
 * then within the implementation of myMethod, the
 * following call can raise a BadCall exception: */

_set_exception(SOMD_SOMOAObject,
     &ev, USER_EXCEPTION, ex_MyObject_BadCall, NULL);
```

Original Class

BOA

Context Class

Description

The **Context** class implements the CORBA Context object described in section 6.5 beginning on page 116 of CORBA 1.1. A **Context** object contains a list of properties, each consisting of a name and a string value associated with that name. **Context** objects are created/accessed by the **get_default_context** method defined in the **ORB** object.

File Stem

cntxt

Base

SOMObject

Metaclass

SOMClass

Ancestor Classes

SOMObject

New Methods

create_child

delete_values

destroy *

get_values

set_one_value

set_values

(* The **destroy** method was defined as **delete** in CORBA 1.1, which conflicts with the **delete** operator in C++. However, there is a **Context_delete** macro defined for CORBA compatibility.)

Overridden Methods

somInit

create_child Method

Purpose

Creates a child of a **Context** object.

IDL Syntax

ORBStatus create_child (
in Identifier *ctx_name***,**
out Context *child_ctx***);**

Description

The **create_child** method creates a child **Context** object.

The returned **Context** object is chained to its parent. That is, searches on the child **Context** object will look in the parent (and so on, up the **Context** tree), if necessary, for matching property names.

Parameters

receiver	A pointer to the **Context** object for which a child is to be created.
env	A pointer to the **Environment** structure for the method caller.
ctx_name	The name of the child **Context** to be created.
child_ctx	The address where a pointer to the created child **Context** object is to be stored.

Return Value

The **create_child** method returns an **ORBStatus** value representing the return code from the operation.

Example

```
#include <somd.h>

Environment ev;
Context cxt, newcxt;
long rc;
...
/* get the process' default Context */
rc = _get_default_context(SOMD_ORBObject, &ev, &cxt);
/* make newcxt a child Context of the default Context (cxt) */
rc = _create_child(cxt, &ev, "myContext", &newcxt);
```

Original Class

Context

delete_values Method

Purpose

Deletes property value(s).

IDL Syntax

ORBStatus delete_values (
in Identifier *prop_name***);**

Description

The **delete_values** method deletes the specified property value(s) from a **Context** object. If *prop_name* has a trailing wildcard character("*"), then all property names that match will be deleted.

Search scope is always limited to the specified **Context** object.

If no matching property is found, an exception is returned.

Parameters

receiver	A pointer to the **Context** object from which values will be deleted.
env	A pointer to the **Environment** structure for the method caller.
prop_name	An identifier specifying the property value(s) to be deleted.

Return Value

The **delete_values** method returns an **ORBStatus** value representing the return code from the operation.

Example

```
#include <somd.h>

Environment ev;
Context cxt, newcxt;
long rc;
...
/* get the process' default Context */
rc = _get_default_context(SOMD_ORBObject, &ev, &cxt);
/* make newcxt a child Context of the default Context (cxt) */
rc = _create_child(cxt, &ev, "myContext", &newcxt);
rc = _set_one_value(newcxt, &ev, "username", "joe");
...
rc = _delete_values(newcxt, &ev, "username");
```

Original Class

Context

Related Information

Methods: set_one_value, **set_values**, **get_values**

destroy Method (for a Context object)

Purpose

Deletes a **Context** object.

IDL Syntax

ORBStatus destroy (
                    in Flags *del_flag***);**

Description

The **destroy** method deletes the specified **Context** object.

NOTE: This method is called “delete” in the CORBA 1.1 specification. However, the word “delete” is a reserved operator in C++, so the name “destroy” was chosen as an alternative. For CORBA compatibility, a macro defining **Context_delete** as an alias for **destroy** has been included in the C header files.

Parameters

receiver	A pointer to the **Context** object to be deleted.
env	A pointer to the **Environment** structure for the method caller.
del_flag	A bitmask (unsigned long). If the option flag CTX_DELETE_DESCENDENTS is specified, the method deletes the indicated **Context** object and all of its descendent **Context** objects. Or, a zero value indicates the flag is not set.

Return Value

The **destroy** method returns an **ORBStatus** value representing the return code from the operation.

Example

```
#include <somd.h>

Environment ev;
Context cxt, newcxt;
long rc;
...
/* get the process’ default Context */
rc = _get_default_context(SOMD_ORBObject, &ev, &cxt);
/* make newcxt a child Context of the default Context (cxt) */
rc = _create_child(cxt, &ev, ”myContext”, &newcxt);
...
/* assuming no descendent Contexts have been
 * created from newcxt, we can destroy newcxt with flags=0
 */
rc = _destroy(newcxt, &ev, (Flags) 0);
```

Original Class

Context

get_values Method

Purpose

Retrieves the specified property values.

IDL Syntax

ORBStatus get_values (
in Identifier *start_scope***,**
in Flags *op_flags***,**
in Identifier *prop_name***,**
out NVList *values***);**

Description

The **get_values** method retrieves the specified **Context** property values(s). If *prop_name* has a trailing wildcard character("*"), then all matching properties and their values are returned. OWNERSHIP of the returned **NVList** object is transferred to the caller.

If no properties are found, an error is returned and no property list is returned.

Scope indicates the level at which to initiate the search for the specified properties. If a property is not found at the indicated level, the search continues up the **Context** object tree until a match is found or all **Context** objects in the chain have been exhausted.

If scope name is omitted, the search begins with the specified **Context** object. If the specified scope name is not found, an exception is returned.

Parameters

receiver	A pointer to the **Context** object from which the properties are to be retrieved.
env	A pointer to the **Environment** structure for the method caller.
start_scope	An **Identifier** specifying the name of the **Context** object at which search for the properties should commence.
op_flags	A bitmask (**long**). The operation flag CTX_RESTRICT_SCOPE may be specified. Searching is limited to the specified search scope or **Context** object.
prop_name	An **Identifier** specifying the name of the property value(s) to return.
values	The address to store a pointer to the resulting **NVList** object.

Return Value

The **get_values** method returns an **ORBStatus** value representing the return code from the operation.

Example

```
#include <somd.h>

Environment ev;
Context cxt1, cxt2;
string *cxt1props;
long rc, i, numprops;
NVList nvp;
...
for (i= numprops; i > 0; i--) {
 /* get the value of the *cxt1props property from cxt1 */
 rc = _get_values(cxt1, &ev, NULL, (Flags) 0, *cxt1props, &nvp);
 /* and if found then update cxt2 with that name-value pair */
 if (rc == 0) rc = _set_values(cxt2, &ev, nvp);
 _free(nvp,&ev);
 cxt1props++;
}
```

Original Class

Context

Related Information

Methods: set_one_value, **set_values**, **delete_values**

set_one_value Method

Purpose

Adds a single property to the specified **Context** object.

IDL Syntax

ORBStatus set_one_value (
in Identifier *prop_name*,
in string *value*);

Description

The **set_one_value** method adds a single property to the specified **Context** object.

Parameters

receiver	A pointer to the **Context** object to which the value is to be added.
env	A pointer to the **Environment** structure for the method caller.
prop_name	The name of the property to be added. The *prop_name* should not end in an asterisk.
value	The value of the property to be added.

Return Value

The **set_one_value** method returns an **ORBStatus** value representing the return code from the operation.

Example

```
#include <somd.h>

Environment ev;
Context cxt, newcxt;
long rc;
...
/* get the process' default Context */
rc = _get_default_context(SOMD_ORBObject, &ev, &cxt);
/* make newcxt a child Context of the default Context (cxt) */
rc = _create_child(cxt, &ev, "myContext", &newcxt);
rc = _set_one_value(newcxt, &ev, "username", "joe");
```

Original Class

Context

Related Information

Methods: set_values, **get_values**, **delete_values**

set_values Method

Purpose

Adds/changes one or more property values in the specified **Context** object.

IDL Syntax

ORBStatus set_values (
in NVList *values***);**

Description

The **set_values** method sets one or more property values in the specified **Context** object. In the **NVList**, the flags field must be set to zero, and the **TypeCode** field associated with an attribute value must be **TC_string**.

Parameters

receiver	A pointer to the **Context** object for which the properties are to be set.
env	A pointer to the **Environment** structure for the method caller.
values	A pointer to an **NVList** object containing the properties to be set. The property names in the **NVList** should not end in an asterisk.

Return Value

The **set_values** method returns an **ORBStatus** value representing the return code from the operation.

Example

```
#include <somd.h>

Environment ev;
Context cxt1, cxt2;
string *cxt1props;
long rc, i, numprops;
NVList nvp;
...
for (i= numprops; i > 0; i--) {
 /* get the value of the *cxt1props property from cxt1 */
 rc = _get_values(cxt1, &ev, NULL, (Flags) 0, *cxt1props, &nvp);
 /* and if found then update cxt2 with that name-value pair */
 if (rc == 0) rc = _set_values(cxt2, &ev, nvp);
 _free(nvp,&ev);
 cxt1props++;
}
```

Original Class

Context

Related Information

Methods: set_one_value, **get_values**, **delete_values**

ImplementationDef Class

Description

The **ImplementationDef** class defines attributes necessary for the DSOM daemon to find and activate the implementation of an object.

Note: Details of the **ImplementationDef** object are not currently defined in the CORBA 1.1 specification; the attributes which have been defined are required by DSOM.

File Stem

impldef

Base

SOMObject

Metaclass

SOMClass

Ancestor Classes

SOMObject

Attributes

The following is a list of each available attribute, with its corresponding type in parentheses, followed by a description of its purpose:

impl_id (string)
Contains the DSOM-generated identifier for a server implementation.

impl_alias (string)
Contains the “alias” (user-friendly name) for a server implementation.

impl_program (string)
Contains the name of the program or command file which will be executed when a process for this server is started automatically by **somdd**. If the full pathname is not specified, the directories specified in the PATH environment variable will be searched for the named program or command file.

Optionally, the server program can be run under control of a “shell” or debugger, by specifying the shell or debugger name first, followed by the name of the server program. (A space separates the two program names.) For example,

```
dbx myserver
```

Servers that are started automatically by **somdd** will always be passed their **impl_id** as the first parameter.

impl_flags (Flags)
Contains a bit-vector of flags used to identify server options. Currently, the IMPLDEF_MULTI_THREAD flag indicates that each request should be executed on a separate thread (OS/2 only). IMPLDEF_DISABLE_SVR indicates that the server process has been disabled from starting.

impl_server_class (string)
Contains the name of the **SOMDServer** class or subclass created by the server process.

impl_refdata_file (string)
Contains the full pathname of the file used to store **ReferenceData** for the server.

impl_refdata_bkup (string)
Contains the full pathname of the backup mirror file used to store **ReferenceData** for the server.

impl_hostname (string)
Contains the hostname of the machine where the server is located.

Notes

Currently, when stored in the Implementation Repository, file names used in **ImplementationDef**s are limited to 255 bytes. Implementations aliases used in **ImplementationDef**s are limited to 50 bytes. Class names used in **ImplementationDef**s are limited to 50 bytes. Hostnames are limited to 32 bytes.

ImplRepository Class

Description

The **ImplRepository** class defines operations necessary to query and update the DSOM Implementation Repository.

Note: The Implementation Repository is described in concept in the CORBA 1.1 specification, but no standard interfaces have been defined. These interfaces have all been introduced by DSOM. In addition to using the following interfaces, the DSOM Implementation Repository can be queried and updated using the **regimpl** tool.

File Stem

implrep

Base

SOMObject

Metaclass

SOMMSingleInstance

Ancestor Classes

SOMObject

New Methods

add_class_to_impldef

add_impldef

delete_impldef

find_all_impldefs

find_classes_by_impldef

find_impldef

find_impldef_by_alias

find_impldef_by_class

remove_class_from_all

remove_class_from_impldef

update_impldef

Overridden Methods

somInit

somUninit

add_class_to_impldef Method

Purpose

Associates a class with a server.

IDL Syntax

void add_class_to_impldef (
in ImplId *implid*,
in string *classname* **);**

Description

Associates a class, identified by name, with a server, identified by its **ImplId**. This type of association is used to lookup server implementations via the **find_impldef_by_class** method.

Parameters

receiver	A pointer to the **ImplRepository** object.
env	A pointer to the **Environment** structure for the method caller.
implid	The **ImplId** identifier for the **ImplementationDef** of the desired server.
classname	A **string** identifying the class name.

Return Value

An exception is returned if there was an error updating the Implementation Repository.

Example

```
#include <somd.h>

Environment ev;
SOMDServer server;
ImplementationDef impldef;
ImplId implid;
...
server = _somdFindServerByName(SOMD_ObjectMgr,&ev,"stackServer");
impldef = _get_implementation(server,&ev);
implid = __get_impl_id(impldef,&ev);
_add_class_to_impldef(SOMD_ImplRepObject,&ev,implid,"Queue");
```

Original Class

ImplRepository

add_impldef Method

Purpose

Adds an implementation definition to the Implementation Repository.

IDL Syntax

void add_impldef (
    in ImplementationDef *impldef***);**

Description

Adds the specified **ImplementationDef** object to the Implementation Repository.

Note: the **impl_id** field of the **ImplementationDef** is ignored. A new **impl_id** value will be created for the newly added **ImplementationDef.**

Parameters

receiver	A pointer to the **ImplRepository** object.
env	A pointer to the **Environment** structure for the method caller.
impldef	A pointer to the **ImplementationDef** object to add to the Implementation Repository.

Return Value

An exception is returned if there was an error updating the Implementation Repository.

Example

```
#include <somd.h>

Environment ev;
ImplementationDef impldef;
...
impldef = ImplementationDefNew();
__set_impl_program(impldef,&ev,"/u/servers/myserver");
/* set more of the impldef's attributes here */
...
_add_impldef(SOMD_ImplRepObject,&ev,impldef);
```

Original Class

ImplRepository

delete_impldef Method

Purpose

Deletes an implementation definition from the Implementation Repository.

IDL Syntax

void delete_impldef (
in ImplId *implid* **);**

Description

Deletes the specified **ImplementationDef** object from the Implementation Repository.

Parameters

receiver	A pointer to the **ImplRepository** object.
env	A pointer to the **Environment** structure for the method caller.
implid	The **ImplId** that identifies the server implementation of interest.

Return Value

An exception is returned if there was an error updating the Implementation Repository.

Example

```
#include <somd.h>

Environment ev;
ImplementationDef impldef;
...
impldef =
 _find_impldef_by_name(SOMD_ImplRepObject,&ev,"stackServer");
_delete_impldef(SOMD_ImplRepObject,&ev,__get_impl_id(impldef,&ev));
```

Original Class

ImplRepository

find_all_impldefs Method

Purpose

Returns all the implementation definitions in the Implementation Repository.

Syntax

ORBStatus find_all_impldefs (out sequence<ImplementationDef> *outimpldefs***);**

Description

The **find_all_impldefs** method searches the Implementation Repository and returns all the **ImplementationDef** objects in it.

Parameters

receiver	A pointer to an object of class **ImplRepository**.
ev	A pointer to the **Environment** structure for the calling method.
outimpldefs	A sequence of **ImplementationDefs** is returned.

Return Value

A zero is returned to indicate success; otherwise, a DSOM error code is returned.

Example

```
#include <somd.h>

Environment ev;
sequence (ImplementationDef) impldefs;

. . .

find_all_impldefs(SOMD_ImplRepObject, &ev, &impldefs);
```

Original Class

ImplRepository

find_classes_by_impldef Method

Purpose

Returns a sequence of class names associated with a server.

IDL Syntax

sequence<string> find_classes_by_impldef (
in ImplId *implid* **);**

Description

The **find_classes_by_impldef** method searches the class index and returns the sequence of class names supported by a server with the specified *implid*.

Parameters

receiver	A pointer to the **ImplRepository** object.
env	A pointer to the **Environment** structure for the method caller.
implid	The **ImplId** that identifies the server implementation of interest.

Return Value

A sequence of strings is returned. *Ownership* of the sequence structure, the string array buffer, and the strings themselves is transferred to the caller.

An exception is returned if there was an error reading the Implementation Repository.

Example

```
#include <somd.h>

Environment ev;
SOMDServer server;
ImplementationDef impldef;
ImplId implid;
sequence(string) classes;
...
server = _find_server_by_name(SOMD_ObjectMgr,&ev,”stackServer”);
impldef = _get_implementation(server,&ev);
implid = __get_impl_id(impldef,&ev);
classes = _find_classes_by_impldef(SOMD_ImplRepObject,&ev,implid);
```

Original Class

ImplRepository

find_impldef Method

Purpose

Returns a server implementation definition given its ID.

IDL Syntax

ImplementationDef find_impldef (
in ImplId *implid***);**

Description

Finds and returns the **ImplementationDef** object whose ID is *implid*.

Parameters

receiver	A pointer to the **ImplRepository** object.
env	A pointer to the **Environment** structure for the method caller.
implid	The **ImplId** of the desired **ImplementationDef.**

Return Value

A copy of the desired **ImplementationDef** object is returned. *Ownership* of the object is transferred to the caller.

An exception is returned if there was an error reading the Implementation Repository.

Example

```
#include <somd.h>

main(int argc, char **argv)
{
 Environment ev;
 SOM_InitEnvironment(&ev);

 /* Initialize the DSOM run-time environment */
 SOMD_Init(&ev);

 /* Retrieve its ImplementationDef from the Implementation
  Repository by passing its implementation ID as a key */
 SOMD_ImplDefObject =
  _find_impldef(SOMD_ImplRepObject, &ev, argv[1]);

 /* Tell DSOM that the server is ready to process requests */
 _impl_is_ready(SOMD_SOMOAObject, &ev, SOMD_ImplDefObject);
 ...
}
```

Original Class

ImplRepository

find_impldef_by_alias Method

Purpose

Returns a server implementation definition, given its user-friendly alias.

IDL Syntax

ImplementationDef find_impldef_by_alias (
in string *alias_name***);**

Description

Finds and returns the **ImplementationDef** object whose alias is *alias_name.*

Parameters

receiver	A pointer to the **ImplRepository** object.
env	A pointer to the **Environment** structure for the method caller.
alias_name	User-friendly name used to identify the implementation.

Return Value

A copy of the desired **ImplementationDef** object is returned, and *ownership* of the object is transferred to the caller. Or, if the specified alias is not found in the Implementation Repository, NULL is returned.

An exception is returned if there was an error reading the Implementation Repository.

Example

```
#include <somd.h>

Environment ev;
ImplementationDef impldef;
...
impldef =
 _find_impldef_by_name(SOMD_ImplRepObject,&ev,"stackServer");
_delete_impldef(SOMD_ImplRepObject,&ev,__get_impl_id(impldef,&ev));
```

Original Class

ImplRepository

find_impldef_by_class Method

Purpose

Returns a sequence of implementation definitions for servers that are associated with a specified class.

IDL Syntax

sequence<ImplementationDef> find_impldef_by_class (
in string *classname***);**

Description

Returns a sequence of **ImplementationDefs** for those servers that have registered an association with a specified class. Typically, a server will be associated with the classes it knows how to implement by registering its known classes via the **add_class_to_impldef** method.

Parameters

receiver	A pointer to the **ImplRepository** object.
env	A pointer to the **Environment** structure for the method caller.
classname	A **string** whose value is the class name of interest.

Return Value

Copies of all **ImplementationDef** objects are returned in a sequence. *Ownership* of the sequence structure, the object array buffer, and the objects themselves is transferred to the caller.

An exception is returned if there was an error reading the Implementation Repository.

Example

```
#include <somd.h>

Environment ev;
sequence(ImplementationDef) impldefs;
...
impldefs =
 _find_impldef_by_class(SOMD_ImplRepObject,&ev,”Stack”);
```

Original Class

ImplRepository

remove_class_from_all Method

Purpose

Removes the association of a particular class from all servers.

Syntax

void remove_class_from_all (in string *className***);**

Description

The **remove_class_from_all** method removes the *className* from all of the **ImplementationDefs**.

Parameters

receiver	A pointer to an object of class **ImplRepository**.
ev	A pointer to the **Environment** structure for the calling method.
className	A string whose value is the class name of interest.

Example

```
#include <somd.h>

Environment ev;
...
remove_class_from_all(SOMD_ImplRepObject, &ev, ”Stack”);
```

Original Class

ImplRepository

remove_class_from_impldef Method

Purpose

Removes the association of a particular class with a server.

IDL Syntax

void remove_class_from_impldef (
in ImplId *implid*,
in string *classname* **);**

Description

Removes the specified class name from the set of class names associated with the server implementation identified by *implid.*

Parameters

receiver	A pointer to the **ImplRepository** object.
env	A pointer to the **Environment** structure for the method caller.
implid	A pointer to an **ImplRepository** object.
classname	A **string** whose value is the class name of interest.

Return Value

An exception is returned if there was an error updating the Implementation Repository.

Example

```
#include <somd.h>

Environment ev;
SOMDServer server;
ImplementationDef impldef;
ImplId implid;
...
server = _find_server_by_name(SOMD_ObjectMgr,&ev,"stackServer");
impldef = _get_implementation(server,&ev);
implid = __get_impl_id(impldef,&ev);
_remove_class_from_impldef(SOMD_ImplRepObject,
         &ev,implid,"Queue");
```

Original Class

ImplRepository

update_impldef Method

Purpose

Updates an implementation definition in the Implementation Repository.

IDL Syntax

void update_impldef (
in ImplementationDef *impldef***);**

Description

Replaces the state of the specified **ImplementationDef** object in the Implementation Repository. The ID of the *impldef* determines which object gets updated in the Implementation Repository.

Parameters

receiver	A pointer to the **ImplRepository** object.
env	A pointer to the **Environment** structure for the method caller.
impldef	A pointer to an **ImplementationDef** object, whose values are to be saved in the Implementation Repository.

Return Value

An exception is returned if there was an error updating the Implementation Repository.

Example

```
#include <somd.h>

Environment ev;
SOMDObject objref;
ImplementationDef impldef;
...
impldef = _get_implementation(objref,&ev);
__set_impl_program(impldef,&ev,"/u/joe/bin/myserver");
_update_impldef(SOMD_ImplRepObject,&ev,impldef);
```

Original Class

ImplRepository

NVList Class

Description

The type **NamedValue** is a standard datatype defined in CORBA (see the CORBA 1.1 page 106). It can be used either as a parameter type or as a mechanism for describing arguments to a request. The **NVList** class implements the **NVList** object used for constructing lists composed of **NamedValues**. **NVLists** can be used to describe arguments passed to request operations or to pass lists of property names and values to context object routines. Additional information about **NVList** is contained in Chapter 6 of the CORBA 1.1 specification.

File Stem

nvlist

Base

SOMObject

Metaclass

SOMClass

Ancestor Classes

SOMObject

New Methods

add_item

free

free_memory

get_count

get_item *

set_item *

(* These methods were added by DSOM to supplement the published CORBA 1.1 methods.)

Overridden Methods

somInit

add_item Method

Purpose

Adds an item to the specified **NVList**.

IDL Syntax

ORBStatus add_item (
in Identifier *item_name*,
in TypeCode *item_type*,
in void* *value*,
in long *value_len*,
in Flags *item_flags*);

Description

The **add_item** method adds an item to the end of the specified list.

Parameters

receiver	A pointer to the **NVList** object to which the item will be added.
env	A pointer to the **Environment** structure for the method caller.
item_name	The name of the item to be added.
item_type	The data type of the item to be added.
value	A pointer to the value of the item to be added.
value_len	The length of the item value to be added.
item_flags	A **Flags** bitmask (unsigned long). The *item_flags* can be one of the following values to indicate parameter direction:

ARG_IN	The argument is input only.
ARG_OUT	The argument is output only.
ARG_INOUT	The argument is input/output.

In addition, item_flags may also contain the following values:

IN_COPY_VALUE	An internal copy of the argument is made and used.
DEPENDENT_LIST	Indicates that a specified sublist must be freed when the parent list is freed.

Return Value

The **add_item** method returns an **ORBStatus** value representing the return code from the operation.

Example

```
#include <somd.h>

Environment ev;
NVList plist;
ORBStatus rc;
...
rc = _create_list(SOMD_ORBObject, &ev, 0, &plist);
rc = _add_item(plist, &ev, ”firstname”, TC_string, ”Joe”, 3, 0);
rc = _add_item(plist, &ev, ”lastname”, TC_string, ”Schmoe”, 5, 0);
```

Original Class

NVList

Related Information

Methods: free, **free_memory**, **get_count**, **get_item**, **set_item**, **create_list**

free Method

Purpose

Frees a specified **NVList.**

IDL Syntax

ORBStatus free ( );

Description

The **free** method frees an **NVList** object and any associated memory. It makes an implicit call to the **free_memory** method.

Parameters

receiver	A pointer to the **NVList** object to be freed.
env	A pointer to the **Environment** structure for the method caller.

Return Value

The method returns an **ORBStatus** value representing the return code from the operation.

Example

```
#include <somd.h>

Environment ev;
long nargs;
NVList arglist;
ORBStatus rc;
...
rc = _create_list(SOMD_ORBObject, &ev, nargs, &arglist);
...
rc= _free(arglist,&ev);
```

Original Class

NVList

Related Information

Methods: free_memory

Functions: ORBfree

free_memory Method

Purpose

Frees any dynamically allocated out-arg memory associated with the specified list.

IDL Syntax

ORBStatus free_memory ( );

Description

The **free_memory** method frees any dynamically allocated out-arg memory associated with the specified list, without freeing the list object itself. This would be useful when invoking a DII request multiple times with the same **NVList.**

Parameters

receiver	A pointer to the **NVList** object whose out-arg memory is to be freed.
env	A pointer to the **Environment** structure for the method caller.

Return Value

The **free_memory** method returns an **ORBStatus** value representing the return code from the operation.

Example

```
#include <somd.h>
#include <repostry.h>
#include <intfacdf.h>
#include <foo.h> /* provided by user */

/* assume following method declaration in interface Foo:
 *      long methodLong  (in long inLong,inout long inoutLong);
 * then the following code repeatedly invokes a request:
 *      result = methodLong(fooObj, &ev, 100, 200);
 * using the DII.
 */

Environment ev;
NVList arglist;
NamedValue result;
long rc;
Foo fooObj;
Request reqObj;

/* See example code for ”invoke” to see how the argList is built */

/* Create the Request, reqObj */
rc = _create_request(fooObj, &ev, (Context *)NULL, ”methodLong”,
       arglist, &result, &reqObj, (Flags)0);

/* Repeatedly invoke the Request */
for (;;) {
 rc = _invoke(reqObj, &ev, (Flags)0);
 ...
 rc= _free_memory(arglist,&ev);  /* free out args */
}
...
```

Original Class

NVList

Related Information

Methods: free

Functions: ORBfree

get_count Method

Purpose

Returns the total number of items allocated for a list.

IDL Syntax

ORBStatus get_count (
out long *count***);**

Description

The **get_count** method returns the total number of allocated items in the specified list.

Parameters

receiver	A pointer to the **NVList** object on which count is desired.
env	A pointer to the **Environment** structure for the method caller.
count	A pointer to where the method will store the **long** integer count value.

Return Value

The **get_count** method returns an **ORBStatus** value representing the return code from the operation.

Example

```
#include <somd.h>

Environment ev;
long nargs, list_size;
NVList arglist;
ORBStatus rc;
...
rc = _create_list(SOMD_ORBObject, &ev, nargs, &arglist);
...
rc = _get_count(arglist,&ev,&list_size);
```

Original Class

NVList

Related Information

Methods: add_item, **get_item**, **set_item**, **create_list**

get_item Method

Purpose

Returns the contents of a specified list item.

IDL Syntax

ORBStatus get_item (
    in long *item_number*,
    out Identifier *item_name*,
    out TypeCode *item_type*,
    out void* *value*,
    out long *value_len*,
    out Flags *item_flags*);

Description

The **get_item** method gets an item from the specified list. Items are numbered from 0 through *N*. The mode flags can be one of the following values:

The **get_item** method transfers ownership of storage allocated for the item value to the caller.

Parameters

receiver	A pointer to an **NVList** object.
env	A pointer to the **Environment** structure for the method caller.
item_number	The position (index) of the item in the list. The item_number ranges from 0 to *n*–1, where *n* is the total number of items in the list.
item_name	A pointer to where the name of the item should be returned.
item_type	A pointer to where the data type of the item should be returned.
value	A pointer to where a pointer to the value of the item should be returned.
value_len	A pointer to where the length of the item value should be returned.
item_flags	A **Flags** bitmask (unsigned long). The *item_flags* can be one of the following values indicating parameter direction.

ARG_IN	The argument is input only.
ARG_OUT	The argument is output only.
ARG_INOUT	The argument is input/output.

In addition, *item_flags* can have the following values:

IN_COPY_VALUE	Indicates a copy of the argument is contained and used by the **NVList**.
DEPENDENT_LIST	Indicates that a specified sublist must be freed when the parent list is freed.

Return Value

The **get_item** method returns 0 for success, or a DSOM error code for failure (often because item_number+1 exceeds the number of items in the list).

Example

```
#include <somd.h>

Environment ev;
long i, nArgs;
ORBStatus rc;
Identifier name;
TypeCode typeCode;
void *value;
long len;
Flags flags;
NVList argList;
...
/* get number of args */
rc = _get_count(argList, ev, &nArgs);
for (i = 0; i < nArgs; i++) {
 /* get item description */
 rc = _get_item(argList,
      &ev,
      i,
      &name,
      &typeCode,
      &value,
      &len,
      &flags);
 ...
 }
```

Original Class

NVList

Related Information

Methods: add_item, **set_item**, **create_list**

set_item Method

Purpose

Sets the contents of an item in a list.

IDL Syntax

ORBStatus set_item (
    in long *item_number*,
    in Identifier *item_name*,
    in TypeCode *item_type*,
    in void* *value*,
    in long *value_len*,
    in Flags *item_flags*);

Description

The **set_item** method sets the contents of an item in the list.

Parameters

receiver	A pointer to an **NVList** which contains the item to be set.
env	A pointer to the **Environment** structure for the method caller.
item_number	The position (index) of the item in the list. The *item_number* ranges from 0 to *n*–1, where *n* is the total number of items in the list.
item_name	The name of the set item.
item_type	The data type of the set item.
value	A pointer to the value of the set item.
value_len	The length of the set item value.
item_flags	A **Flags** bitmask (unsigned long). The *item_flags* can be one of the following values to indicate parameter direction:

ARG_IN	The argument is input only.
ARG_OUT	The argument is output only.
ARG_INOUT	The argument is input/output.

In addition, *item_flags* may also contain the following values:

IN_COPY_VALUE	Indicates an internal copy of the argument is made and used.
DEPENDENT_LIST	Indicates that a specified sublist must be freed when the parent list is freed.

Return Value

The **set_item** method returns 0 on successful completion or a DSOM error code upon failure (often because item_number+1 exceeds the number of items in the list).

Example

```
#include <somd.h>

Environment ev;
long i, nArgs;
ORBStatus rc;
Identifier name;
TypeCode typeCode;
void *value;
long len;
Flags flags;
NVList argList;
...
/* get number of args */
rc = _get_count(argList, ev, &nArgs);
for (i = 0; i < nArgs; i++) {
 /* change item description */
 rc = _set_item(argList,
       &ev,
       i,
       name,
       typeCode,
       value,
       len,
       flags);
 ...
 }
```

Original Class

NVList

Related Information

Methods: add_item, **get_item**, **create_list**

ObjectMgr Class

Description

The **ObjectMgr** class provides a uniform, universal abstraction for any sort of object manager. Object Request Brokers, persistent storage managers, and OODBMSs are examples of object managers.

This is an abstract base class, which defines the “core” interface for an object manager. It provides basic methods that:

- Create a new object of a certain class,
- Return a (persistent) ID for an object,
- Return a reference to an object associated with an ID,
- Free an object (that is, release any local memory associated with the object without necessarily destroying the object itself), or
- Destroy an object.

Note: The **ObjectMgr** is an *abstract* class and should not be instantiated. Any subclass of **ObjectMgr** must provide implementations for all **ObjectMgr** methods. In DSOM, the class **SOMDObjectMgr** provides a DSOM-specific implementation.

File Stem

om

Base

SOMObject

Metaclass

SOMMSingleInstance

Ancestor Classes

SOMObject

Subclasses

SOMDObjectMgr

New Methods

somdDestroyObject Method *

somdGetIdFromObject Method *

somdGetObjectFromId Method *

somdNewObject Method *

somdReleaseObject Method *

(* This class and its methods were added by DSOM to supplement the published CORBA 1.1 interfaces.)

somdDestroyObject Method

Purpose

Requests destruction of the target object.

IDL Syntax

void somdDestroyObject (
in SOMObject *obj***);**

Description

The **somdDestroyObject** method indicates that the object manager should destroy the specified object. Storage associated with the object is freed.

In DSOM, the **SOMDObjectMgr** forwards the deletion request to the remote server, and then frees the local proxy object.

Parameters

receiver	A pointer to an **ObjectMgr** object.
env	A pointer to the **Environment** structure for the method caller.
obj	A pointer to the object to be freed.

Example

```
#include <somd.h>

Stack stk;
Environment ev;
SOMDServer server;

SOM_InitEnvironment(&ev);
SOMD_Init(&ev);
StackNewClass(0,0);
server =
  _somdFindAnyServerByClass(SOMD_ObjectMgr, &ev,"Stack");
stk = _somdCreateObj(server, &ev, "Stack", "");
...
_somdDestroyObject(SOMD_ObjectMgr, &ev, stk);
```

Original Class

ObjectMgr

Related Information

Methods: somdReleaseObject, **somdCreateObj**, **somdTargetFree**, **release**

somdGetIdFromObject Method

Purpose

Returns an ID for an object managed by a specified Object Manager.

IDL Syntax

string somdGetIdFromObject (
in SOMObject *obj***);**

Description

The **somdGetIdFromObject** method returns the persistent ID for an object managed by the specified Object Manager. This ID is unambiguous — it always refers to the same object.

The **somdGetIdFromObject** method transfers *ownership* of storage allocated for the string to the caller.

Parameters

receiver	A pointer to an **ObjectMgr** object.
env	A pointer to the **Environment** structure for the method caller.
obj	A pointer to the object for which an ID is needed.

Return Value

The **somdGetIdFromObject** method returns a string representing the ID of the specified object.

Example

```
#include <somd.h>
#include <car.h>

Environment ev;
Car car;
string somdObjectId;
/*note that ”SOMDObject Identifiers” are just strings */

SOM_InitEnvironment(&ev);
SOMD_Init(&ev);

/* create a remote Car object */
car = _somdNewObject(SOMD_ObjectMgr, &ev, “Car”, ””);

/* save the reference to the object */
somdObjectId = _somdGetIdFromObject(SOMD_ObjectMgr, &ev, car);
FileWrite(”/u/joe/mycar”, somdObjectId);
...
```

Original Class

ObjectMgr

Related Information

Methods: somdGetObjectFromId

somdGetObjectFromId Method

Purpose

Finds and activates an object implemented by a specified object manager, given its ID.

IDL Syntax

SOMObject somdGetObjectFromId (
in string ***id*****);**

Description

The **somdGetObjectFromId** method finds and activates an object implemented by this object manager, given its ID.

The **somdGetObjectFromId** method transfers *ownership* to the caller.

Parameters

receiver	A pointer to an **ObjectMgr** object.
env	A pointer to the **Environment** structure for the method caller.
id	A string representing an object ID.

Return Value

The **somdGetObjectFromId** method returns a pointer to the object with the specified ID.

Example

```
#include <somd.h>
#include <car.h>
Environment ev;
Car car;
string somdObjectId;
...
/* restore proxy from its string form */
FileRead(”/u/joe/mycar”, &somdObjectId);
car = _somdGetObjectFromId(SOMD_ObjectMgr, &ev, somdObjectId);
...
```

Original Class

ObjectMgr

Related Information

Methods: somdGetIdFromObject

somdNewObject Method

Purpose

Returns a new object of the named class.

IDL Syntax

SOMObject somdNewObject (
in Identifier *objclass***,**
in string *hints***);**

Description

The **somdNewObject** method returns a new object of the class specified by *objclass*. Application-specific creation options can be supplied via the *hints* parameter.

In DSOM, the **SOMDObjectMgr** selects a random server which has advertised knowledge of the desired class *objclass*, and forwards the creation request to that server. The *hints* field is currently ignored by the **SOMDObjectMgr**.

Parameters

receiver	A pointer to an **ObjectMgr** object.
env	A pointer to the **Environment** structure for the method caller.
objclass	An **Identifier** representing the type of the new object.
hints	A **string** which may optionally be used to specify special creation options.

Return Value

The **somdNewObject** method returns a **SOMObject.** *Ownership* of the new object is transferred to the caller.

Example

```
#include <somd.h>
#include <stack.h> /* provided by user */

Stack stk;
Environment ev;
SOMDServer server;

SOM_InitEnvironment(&ev);
SOMD_Init(&ev);
StackNewClass(0,0);
stk = _somdNewObject(SOMD_ObjectMgr, &ev, ”Stack”, ””);
...
_somdDestroyObject(SOMD_ObjectMgr, &ev, stk);
```

Original Class

ObjectMgr

Related Information

Methods: somdDestroyObject, **somdReleaseObject**

somdReleaseObject Method

Purpose

Indicates that the client has finished using the object.

IDL Syntax

void somdReleaseObject (
in SOMObject *obj***);**

Description

The **somdReleaseObject** method indicates that the client has finished using the specified object. This allows the object manager to free the bookkeeping information associated with the object, if any. The object may also be passivated, but it is not destroyed.

In DSOM, **somdReleaseObject** causes the client's proxy for the target object of interest to be freed; the target object is not freed.

Parameters

receiver	A pointer to an **ObjectMgr** object.
env	A pointer to the **Environment** structure for the method caller.
obj	A pointer to the object to be released.

Example

```
#include <somd.h>
#include <car.h>

Environment ev;
Car car;
string somdObjectId;
...
/* restore proxy from its string form */
FileRead("/u/joe/mycar", &somdObjectId);
car = _somdGetObjectFromId(SOMD_ObjectMgr, &ev, somdObjectId);
...
_somdReleaseObject(SOMD_ObjectMgr, &ev, car);
```

Original Class

ObjectMgr

Related Information

Methods: somdDestroyObject, **somdNewObject**, **somdTargetFree**, **release**

ORB Class

Description

The **ORB** class implements the CORBA ORB object described in Chapter 8 of the CORBA 1.1 specification. The **ORB** class defines operations for converting object references to strings and converting strings to object references. The **ORB** also defines operations used by the Dynamic Invocation Interface for creating lists (NVlists) and determining the default context.

File Stem

orb

Base

SOMObject

Metaclass

SOMMSingleInstance

Ancestor Classes

SOMObject

New Methods

create_list

create_operation_list

get_default_context

object_to_string

string_to_object

create_list Method

Purpose

Creates an **NVList** of the specified size.

IDL Syntax

ORBStatus create_list (
in long *count*,
out NVList *new_list***);**

Description

Creates an **NVList** list of the specified size, typically for use in **Request**s.

Ownership of the allocated *new_list* is transferred to the caller.

Parameters

receiver	A pointer to the **ORB** object.
env	A pointer to the **Environment** structure for the method caller.
count	An integer representing the number of elements to allocate for the list.
new_list	A pointer to the address where the method will store a pointer to the allocated **NVList** object.

Return Value

The **create_list** method returns an **ORBStatus** value representing the return code of the operation.

Example

```
#include <somd.h>

Environment ev;
long nargs = 5;
NVList arglist;
ORBStatus rc;
...
rc = _create_list(SOMD_ORBObject, &ev, nargs, &arglist);
```

Original Class

ORB

Related Information

Methods: create_operation_list

create_operation_list Method

Purpose

Creates an **NVList** initialized with the argument descriptions for a given operation.

IDL Syntax

ORBStatus create_operation_list (
in OperationDef *oper*,
out NVList *new_list***);**

Description

Creates an **NVList** list for the specified operation, for use in **Request**s invoking that operation.

Parameters

receiver	A pointer to the **ORB** object.
env	A pointer to the **Environment** structure for the method caller.
oper	A pointer to the **OperationDef** object representing the operation for which the **NVList** is to be initialized.
new_list	A pointer to where the method will store a pointer to the resulting argument list.

Return Value

The **create_operation_list** method returns an **ORBStatus** value representing the return code of the operation.

Ownership of the allocated *new_list* is transferred to the caller.

Example

```
#include <somd.h>

Environment ev;
OperationDef opdef;
NVList arglist;
long rc;

/* Get the OperationDef from the Interface Repository. */
opdef = _lookup_id(SOM_InterfaceRepository,
       &ev, ”Foo:methodLong”);
/* Create a NamedValue list for the operation. */
rc= _create_operation_list(SOMD_ORBObject, &ev, opdef, &arglist);
```

Original Class

ORB

Related Information

Methods: create_list

get_default_context Method

Purpose

Returns the default process **Context** object.

IDL Syntax

ORBStatus get_default_context (
out Context *ctx***);**

Description

The **get_default_context** method gets the default process **Context** object.

Ownership of the allocated **Context** object is transferred to the caller.

Parameters

receiver	A pointer to the **ORB** object.
env	A pointer to the **Environment** structure for the method caller.
ctx	A pointer to where the method will store a pointer to the returned **Context** object.

Return Value

The **get_default_context** method returns an **ORBStatus** return code: 0 indicates success, while a non-zero value is a DSOM error code (see Chapter 6 of the *SOM Toolkit User's Guide*).

Example

```
#include <somd.h>

Environment ev;
Context cxt;
long rc;
...
rc = _get_default_context(SOMD_ORBObject, &ev, &cxt);
```

Original Class

ORB

object_to_string Method

Purpose

Converts an object reference to an external form (string) which can be stored outside the ORB.

IDL Syntax

string object_to_string (
in SOMDObject *obj***);**

Description

The **object_to_string** method converts the object reference to a form (string) which can be stored externally.

Ownership of allocated memory is transferred to the caller.

Parameters

receiver	A pointer to the **ORB** object.
env	A pointer to the **Environment** structure for the method caller.
obj	A pointer to a **SOMDObject** object representing the reference to be converted.

Return Value

The **object_to_string** method returns a string representing the external (string) form of the referenced object.

Example

```
#include <somd.h>
#include <car.h>

Environment ev;
Car car;
string objrefstr;

SOM_InitEnvironment(&ev);
SOMD_Init(&ev);

/* create a remote Car object */
car = _somdNewObject(SOMD_ObjectMgr, &ev, “Car”, ””);

/* save the reference to the object */
objrefstr = _object_to_string(SOMD_ORBObject, &ev, car);
FileWrite(”/u/joe/mycar”, objrefstr);
```

Original Class

ORB

Related Information

Methods: string_to_object

string_to_object Method

Purpose

Converts an externalized (string) form of an object reference into an object reference.

IDL Syntax

SOMDObject string_to_object (
in string *str***);**

Description

The **string_to_object** method converts the externalized (string) form of an object reference into an object reference.

Parameters

receiver	A pointer to the **ORB** object.
env	A pointer to the **Environment** structure for the method caller.
str	A pointer to a character string representing the externalized form of the object reference.

Return Value

The **string_to_object** method returns a **SOMDObject** object.

Example

```
#include <somd.h>
#include <car.h>

Environment ev;
Car car;
string objrefstr;
...
/* restore proxy from its string form */
FileRead(”/u/joe/mycar”, &objrefstr);
car = _string_to_object(SOMD_ORBObject, &ev, objrefstr);
```

Original Class

ORB

Related Information

Methods: object_to_string

Principal Class

Description

The **Principal** class defines attributes which identify the user id and host name of the originator of a specific request. This information is typically used for access control.

A **Principal** object is returned by the **get_principal** method of the SOM Object Adapter. The parameters of the **get_principal** method identify the environment and target object associated with a particular request — the **SOMOA** uses this information to create a **Principal** object which identifies the caller.

Note: Details of the **Principal** object are not currently defined in the CORBA 1.1 specification; the attributes which have been defined are required by DSOM.

File Stem

principl

Base

SOMObject

Metaclass

SOMClass

Ancestor Classes

SOMObject

Attributes

Listed below is each available attribute, with its corresponding type in parentheses, followed by a description of its purpose:

userName (string)
Identifies the name of the user associated with the request invocation. (Currently, this value is obtained from the USER environment variable in the process which invoked the request.)

hostName (string)
Identifies the name of the host from where the request originated. (Currently, this value is obtained from the HOSTNAME environment variable in the process which invoked the request.)

Request Class

Description

The **Request** class implements the CORBA Request object described in section 6.2 on page 108 of CORBA 1.1. The **Request** object is used by the dynamic invocation interface to dynamically create and issue a request to a remote object. **Request** objects are created by the **create_request** method in **SOMDObject**.

File Stem

request

Base

SOMObject

Metaclass

SOMClass

Ancestor Classes

SOMObject

New Methods

add_arg

destroy *

get_response

invoke

send

(* The **destroy** method was defined as **delete** in CORBA 1.1, which conflicts with the **delete** operator in C++. However, there is a **Request_delete** macro defined for CORBA compatibility.)

Overridden Methods

somInit

somUninit

add_arg Method

Purpose

Incrementally adds an argument to a **Request** object.

IDL Syntax

ORBStatus add_arg (
    in Identifier *name*,
    in TypeCode *arg_type*,
    in void* *value*,
    in long *len*,
    in Flags *arg_flags***);**

Description

The **add_arg** method incrementally adds an argument to a **Request** object. The **Request** object must have been created using the **create_request** method with an empty argument list.

Parameters

receiver	A pointer to a **Request** object.
env	A pointer to the **Environment** structure for the method caller.
name	An identifier representing the name of the argument to be added.
arg_type	The typecode for the argument to be added.
value	A pointer to the argument value to be added.
len	The length of the argument.
arg_flags	A **Flags** bitmask (unsigned long). The *arg_flags* parameter may take one of the following values to indicate parameter direction:

ARG_IN	The argument is input only.
ARG_OUT	The argument is output only.
ARG_INOUT	The argument is input/output.

In addition, *arg_flags* may also contain the following values:

IN_COPY_VALUE	An internal copy of the argument is to be made and used.
DEPENDENT_LIST	Indicates that a specified sublist must be freed when the parent list is freed.

Return Value

The **add_arg** method returns an **ORBStatus** value representing the return code of the operation.

Example

```
#include <somd.h>
#include <repostry.h>
#include <intfacdf.h>
#include <foo.h> /* provided by user */

/* assume following method declaration in interface Foo:
 *      long methodLong  (in long inLong,inout long inoutLong);
 * then the following code builds a request to execute the call:
 *      result = methodLong(fooObj, &ev, 100,200);
 *using the DII.
 */

Environment ev;
OperationDef opdef;
Description desc;
OperationDescription *opdesc;
long rc;
long value1 = 100;
long value2 = 200;
Foo fooObj;
Request reqObj;
NamedValue result;

/* Get the OperationDef from the Interface Repository. */
opdef = _lookup_id(SOM_InterfaceRepository,
       &ev, ”Foo::methodLong”);

/* Get the operation description structure. */
desc = _describe(opdef, &ev);
opdesc = (OperationDescription *) desc.value._value;

/* Fill in the TypeCode field for result. */
result.argument._type = opdesc–>result;

/* Create the Request, reqObj */
rc = _create_request(fooObj, &ev, (Context *)NULL, ”methodLong”,
       (NVList *)NULL, &result, &reqObj, (Flags)0);

/* Add arg1 info onto the request */
_add_arg(reqObj, &ev,
    ”inLong”, TC_long, &value1, sizeof(long), (Flags)0);
/* Add arg2 info onto the request */
_add_arg(reqObj, &ev,
    ”inoutLong”, TC_long, &value2, sizeof(long), (Flags)0);
```

Original Class

Request

destroy Method (for a Request object)

Purpose

Deletes the memory allocated by the ORB for a **Request** object.

IDL Syntax

ORBStatus destroy ( );

Description

The **destroy** method deletes the **Request** object and all associated memory.

Note: This method is called “delete” in the CORBA 1.1 specification. However, the word “delete” is a reserved operator in C++, so the name “destroy” was chosen as an alternative. For CORBA compatibility, a macro defining **Request_delete** as an alias for **destroy** has been included in the C header files.

Parameters

receiver	A pointer to a **Request** object.
env	A pointer to the **Environment** structure for the method caller.

Return Value

The **destroy** method returns an **ORBStatus** value representing the return code of the operation.

Example

```
#include <somd.h>
#include <repostry.h>
#include <intfacdf.h>
#include <foo.h> /* provided by user */

/* assume following method declaration in interface Foo:
 *      long methodLong  (in long inLong,inout long inoutLong);
 * then the following code sends a request to execute the call:
 *      result = methodLong(fooObj, &ev, 100,200);
 * using the DII without waiting for the result. Then, later,
 * waits for and then uses the result.
 */
Environment ev;
NVList arglist;
long rc;
Foo fooObj;
Request reqObj;
NamedValue result;

/* see the Example code for invoke to see how the request
 * is built
 */

/* Create the Request, reqObj */
rc = _create_request(fooObj, &ev, (Context *)NULL, "methodLong",
       arglist, &result, &reqObj, (Flags)0);

/* Finally, send the request */
rc = _send(reqObj, &ev, (Flags)0);

/* do some work, i.e. don't wait for the result */

/* wait here for the result of the request */
rc = _get_response(reqObj, &ev, (Flags)0);

/* use the result */
if (result->argument._value == 9600) {...}

/* throw away the reqObj */
_destroy(reqObj, &ev);
```

Original Class

Request

Related Information

Methods: invoke, **send**, **get_response**

get_response Method

Purpose

Determines whether an asynchronous **Request** has completed.

IDL Syntax

ORBStatus get_response (
in Flags *response_flags***);**

Description

The **get_response** method determines whether the asynchronous **Request** has completed.

Parameters

receiver	A pointer to a **Request** object.
env	A pointer to the **Environment** structure for the method caller.
response_flags	A **Flags** bitmask (unsigned long) containing control information for the get_response method. The *response_flags* argument may have the following value: RESP_NO_WAIT Indicates the caller does not want to wait for a response.

Return Value

The **get_response** method returns an **ORBStatus** value representing the return code of the operation.

Example

```
#include <somd.h>
#include <repostry.h>
#include <intfacdf.h>
#include <foo.h> /* provided by user */

/* assume following method declaration in interface Foo:
 *      long methodLong  (in long inLong,inout long inoutLong);
 * then the following code sends a request to execute the call:
 *      result = methodLong(fooObj, &ev, 100,200);
 * using the DII without waiting for the result. Then, later,
 * waits for and then uses the result.
 */

Environment ev;
NVList arglist;
long rc;
Foo fooObj;
Request reqObj;
NamedValue result;

/* see the Example code for invoke to see how the request
 * is built
 */


/* Create the Request, reqObj */
rc = _create_request(fooObj, &ev, (Context *)NULL, "methodLong",
       arglist, &result, &reqObj, (Flags)0);

/* Finally, send the request */
rc = _send(reqObj, &ev, (Flags)0);

/* do some work, i.e. don't wait for the result */

/* wait here for the result of the request */
rc = _get_response(reqObj, &ev, (Flags)0);

/* use the result */
if (result->argument._value == 9600) {...}
```

Original Class

Request

Related Information

Methods: invoke, **send**

Macros: Request_delete

invoke Method

Purpose

Invokes a **Request** synchronously, waiting for the response.

IDL Syntax

ORBStatus invoke (
in Flags *invoke_flags***);**

Description

The **invoke** method sends a **Request** synchronously, waiting for the response.

Parameters

receiver	A pointer to a **Request** object.
env	A pointer to the **Environment** structure for the method caller.
invoke_flags	A **Flags** bitmask (unsigned long) representing control information for the **invoke** method. There are currently no flags defined for the **invoke** method.

Return Value

The **invoke** method returns an **ORBStatus** value representing the return code of the operation.

Example

```
#include <somd.h>
#include <repostry.h>
#include <intfacdf.h>
#include <foo.h> /* provided by user */

/* assume following method declaration in interface Foo:
 *      long methodLong  (in long inLong,inout long inoutLong);
 * then the following code builds and then invokes
 * a request to execute the call:
 *      result = methodLong(fooObj, &ev, 100,200);
 * using the DII.
 */

Environment ev;
OperationDef opdef;
Description desc;
OperationDescription *opdesc;
NVList arglist;
long rc;
long value1 = 100;
long value2 = 200;
Foo fooObj;
Request reqObj;
NamedValue result;
Identifier name;
TypeCode tc;
void *dummy;
long dummylen;
Flags flags;
```

```
/* Get the OperationDef from the Interface Repository. */
opdef = _lookup_id(SOM_InterfaceRepository,
       &ev, ”Foo::methodLong”);
/* Create a NamedValue list for the operation. */
rc= _create_operation_list(SOMD_ORBObject, &ev, opdef, &arglist);

/* Insert arg1 info into arglist */
_get_item(arglist, &ev,
    0, &name, &tc, &dummy, &dummylen, &flags);
_set_item(arglist,&ev,0, name, tc, &value1, sizeof(long), flags);

/* Insert arg2 info into arglist */
_get_item(arglist, &ev,
    1, &name, &tc, &dummy, &dummylen, &flags);
_set_item(arglist,&ev,1, name, tc, &value2, sizeof(long), flags);

/* Get the operation description structure. */
desc = _describe(opdef, &ev);
opdesc = (OperationDescription *) desc.value._value;

/* Fill in the TypeCode field for result. */
result.argument._type = opdesc->result;

/* Create the Request, reqObj */
rc = _create_request(fooObj, &ev, (Context *)NULL, ”methodLong”,
       arglist, &result, &reqObj, (Flags)0);

/* Finally, invoke the request */
rc = _invoke(reqObj, &ev, (Flags)0);

/* Print results */
printf(”result: %d, value2: %d\n”,
   *(long*)(result.argument._value),
   value2);
```

Original Class

Request

Related Information

Methods: send, **get_response**

Macros: Request_delete

send Method

Purpose

Invokes a **Request** asynchronously.

IDL Syntax

ORBStatus send (
    in Flags *invoke_flags***);**

Description

The **send** method invokes the **Request** asynchronously. The response must eventually be checked by invoking either the **get_response** method or the **get_next_response** function.

Parameters

receiver	A pointer to a **Request** object.
env	A pointer to the **Environment** structure for the method caller.
invoke_flags	A **Flags** bitmask (unsigned long) containing **send** method control information. The argument *invoke_flags* can have the following value:

INV_NO_RESPONSE
: Indicates that the invoker does not intend to wait for a response, nor does it expect any of the output arguments (**inout** or **out**) to be updated.

Return Value

The **send** method returns an **ORBStatus** value representing the return code from the operation.

Example

```
#include <somd.h>
#include <repostry.h>
#include <intfacdf.h>
#include <foo.h> /* provided by user */

/* assume following method declaration in interface Foo:
 *      long methodLong  (in long inLong,inout long inoutLong);
 * then the following code sends
 * a request to execute the call:
 *      result = methodLong(fooObj, &ev, 100,200);
 * using the DII.
 */

Environment ev;
NVList arglist;
long rc;
Foo fooObj;
Request reqObj;
NamedValue result;

/* see the Example code for invoke to see how the request
 * is built
 */

/* Create the Request, reqObj */
rc = _create_request(fooObj, &ev, (Context *)NULL, ”methodLong”,
       arglist, &result, &reqObj, (Flags)0);

/* Finally, send the request */
rc = _send(reqObj, &ev, (Flags)0);
```

Original Class

Request

Related Information

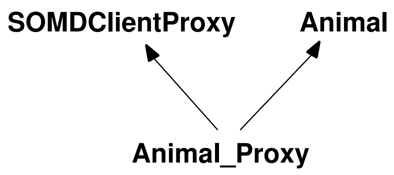
Methods: invoke, **get_response**

Macros: Request_delete

SOMDClientProxy Class

Description

The **SOMDClientProxy** class implements DSOM proxy objects in Clients. **SOMDClientProxy** overrides the usual **somDispatch** methods with versions that build a DSOM **Request** for remote invocation and dispatch it to the remote object. It is intended that the implementation of this “generic” proxy class will be used to derive specific proxy classes via multiple inheritance. The remote dispatch method is inherited from this client proxy class, while the desired interface — and language bindings — are inherited from the target class (but not the implementation).



File Stem

somdcprx

Base

SOMDObject

Metaclass

SOMClass

Ancestor Classes

SOMObject

SOMDObject

New Methods

somdProxyFree *

somdProxyGetClass *

somdProxyGetClassName *

somdReleaseResources *

somdTargetFree *

somdTargetGetClass *

somdTargetGetClassName *

(* This class and its methods were added by DSOM to supplement the published CORBA 1.1 interfaces.)

Overridden methods

create_request

create_request_args

is_proxy

release

somDispatch

somDispatchA, somDispatchD, somDispatchL, somDispatchV

somFree

somGetClass

somGetClassName

somInit

somUninit

somdProxyFree Method

Purpose

Executes **somFree** on the local proxy object.

IDL Syntax

void somdProxyFree ( );

Description

The **somdProxyFree** method executes the **somFree** method call on the local proxy object. This method has been provided when the application program wants to be explicit about freeing the proxy object vs. the target object.

Parameters

receiver	A pointer to the **SOMDClientProxy** object.
env	A pointer to the **Environment** structure for the method caller.

Return Value

somdProxyFree has no return value.

Example

```
#include <somd.h>
#include <car.h>

Environment ev;
Car car;
string somdObjectId;
...
/* restore proxy from its string form */
FileRead(”/u/joe/mycar”, &somdObjectId);
car = _somdGetObjectFromId(SOMD_ObjectMgr, &ev, somdObjectId);
...
_somdProxyFree(car, &ev);
```

Original Class

SOMDClientProxy

Related Information

Methods: release, **somdReleaseObject**

somdProxyGetClass Method

Purpose

Returns the class object for the local proxy object.

IDL Syntax

SOMClass somdProxyGetClass ( );

Description

The **somdProxyGetClass** method executes the **somGetClass** method call on the local proxy object and returns a pointer to the proxy's class object. This method has been provided when the application program wants to be explicit about getting the class object for the proxy object vs. the target object.

Parameters

receiver	A pointer to the **SOMDClientProxy** object.
env	A pointer to the **Environment** structure for the method caller.

Return Value

The **somdProxyGetClass** method returns a pointer to the class object for the local proxy object.

Example

```
#include <somd.h>
#include <car.h>

Environment ev;
Car car;
SOMClass carProxyClass;
string somdObjectId;
...
/* restore proxy from its string form */
FileRead(”/u/joe/mycar”, &somdObjectId);
car = _somdGetObjectFromId(SOMD_ObjectMgr, &ev, somdObjectId);
...
carProxyClass = _somdProxyGetClass(car, &ev);
```

Original Class

SOMDClientProxy

somdProxyGetClassName Method

Purpose

Returns the class name for the local proxy object.

IDL Syntax

string somdProxyGetClassName ( );

Description

The **somdProxyGetClassName** method executes the **somGetClassName** method call on the local proxy object and returns the proxy's class name. This method has been provided when the application program wants to be explicit about getting the class name of the proxy object vs. the target object.

Parameters

receiver	A pointer to the **SOMDClientProxy** object for the desired remote target object.
env	A pointer to the **Environment** structure for the method caller.

Return Value

The **somdProxyGetClassName** method returns a string containing the class name of the local proxy object.

Example

```
#include <somd.h>
#include <car.h>

Environment ev;
Car car;
string carProxyClassName;
string somdObjectId;
...
/* restore proxy from its string form */
FileRead(”/u/joe/mycar”, &somdObjectId);
car = _somdGetObjectFromId(SOMD_ObjectMgr, &ev, somdObjectId);
...
carProxyClassName = _somdProxyGetClassName(car, &ev);
```

Original Class

SOMDClientProxy

somdReleaseResources Method

Purpose

Instructs a proxy object to release any memory it is holding as a result of a remote method invocation in which a parameter or result was designated as “object-owned”.

IDL Syntax

void somdReleaseResources ();

Description

The **somdReleaseResources** method instructs a proxy object to release any memory it is holding as a result of a remote method invocation in which a parameter or result was designated as “object-owned”.

When a DSOM client program makes a remote method invocation, via a proxy, and the method being invoked has an object-owned parameter or return result, the client-side memory associated with the parameter/result will be owned by the caller’s proxy, and the server-side memory will be owned by the remote object. The memory owned by the caller’s proxy will be freed when the proxy is released by the client program. (The time at which the server-side memory will be freed depends on the implementation of the remote object.)

A DSOM client can also instruct a proxy object to free all memory that it owns on behalf of the client without releasing the proxy (assuming that the client program is finished using the object-owned memory), by invoking the **somdReleaseResources** method on the proxy object. Calling **somdReleaseResources** can prevent unused memory from accumulating in a proxy.

For example, consider a client program repeatedly invoking a remote method “get_string”, which returns a string that is designated (in SOM IDL) as “object-owned”. The proxy on which the method is invoked will store the memory associated with all of the returned strings, even if the strings are not unique, until the proxy is released. If the client program only uses the last result returned from “get_string”, then unused memory accumulates in the proxy. The client program can prevent this by invoking **somdReleaseResources** on the proxy object periodically (for example, each time it finishes using the result of the last “get_string” call).

Parameters

receiver	A pointer to the **SOMDClientProxy** object to release resources.
ev	A pointer to the **Environment** structure for the method call.

Example

```
string mystring;
...
/* remote invocation of get_string on proxy x,
 * where method get_string has the SOM IDL modifier
 * ”object_owns_result”.
 */
mystring = X_get_string(x, ev);

/* ... use mystring ... */

/* when finished using mystring, instruct the
 * proxy that it can free it.
 */
_somdReleaseResources(x, ev);
```

Original Class

SOMDClientProxy

Related Information

Methods: release

somdTargetFree Method

Purpose

Forwards the **somFree** method call to the remote target object.

IDL Syntax

void somdTargetFree ( );

Description

The **somdTargetFree** method forwards the **somFree** method call to the remote target object. This method has been provided when the application program wants to be explicit about freeing the remote target object vs. the proxy object.

Parameters

receiver	A pointer to the **SOMDClientProxy** object for the desired remote target object.
env	A pointer to the **Environment** structure for the method caller.

Return Value

somdTargetFree has no return value.

Example

```
#include <somd.h>
#include <car.h>

Environment ev;
Car car;
string somdObjectId;
...
/* restore proxy from its string form */
FileRead(”/u/joe/mycar”, &somdObjectId);
car = _somdGetObjectFromId(SOMD_ObjectMgr, &ev, somdObjectId);
...
_somdTargetFree(car, &ev);
```

Original Class

SOMDClientProxy

Related Information

Methods: release, **somdDestroyObject**

somdTargetGetClass Method

Purpose

Returns (a proxy for) the class object for the remote target object.

IDL Syntax

SOMClass somdTargetGetClass ( );

Description

The **somdTargetGetClass** method forwards the **somGetClass** method call to the remote target object and returns a pointer to the class object for that object. This method has been provided when the application program wants to be explicit about getting the class object for the remote target object vs. the local proxy.

Parameters

receiver	A pointer to the **SOMDClientProxy** object for the desired remote target object.
env	A pointer to the **Environment** structure for the method caller.

Return Value

The **somdTargetGetClass** method returns a pointer to the class object for the remote target object.

Example

```
#include <somd.h>
#include <car.h>

Environment ev;
Car car;
SOMClass carClass;
string somdObjectId;
...
/* restore proxy from its string form */
FileRead(”/u/joe/mycar”, &somdObjectId);
car = _somdGetObjectFromId(SOMD_ObjectMgr, &ev, somdObjectId);
...
carClass = _somdTargetGetClass(car, &ev);
```

Original Class

SOMDClientProxy

Related Information

Methods: somdProxyGetClass

somdTargetGetClassName Method

Purpose

Returns the class name for the remote target object.

IDL Syntax

string somdTargetGetClassName ( );

Description

The **somdTargetGetClassName** method forwards the **somGetClassName** method call to the remote target object and returns the class name for that object. This method has been provided when the application program wants to be explicit about getting the class name of the remote target object vs. the proxy object.

Parameters

receiver	A pointer to the **SOMDClientProxy** object for the desired remote target object.
env	A pointer to the **Environment** structure for the method caller.

Return Value

The **somdTargetGetClassName** method returns a string containing the class name of the remote target object.

Example

```
#include <somd.h>
#include <car.h>

Environment ev;
Car car;
string carClassName;
string somdObjectId;
...
/* restore proxy from its string form */
FileRead(”/u/joe/mycar”, &somdObjectId);
car = _somdGetObjectFromId(SOMD_ObjectMgr, &ev, somdObjectId);
...
carClassName = _somdTargetGetClassName(car, &ev);
```

Original Class

SOMDClientProxy

Related Information

Methods: somdProxyGetClassName

SOMDObject Class

Description

The **SOMDObject** class implements the methods that can be applied to all CORBA object references: for example, **get_implementation**, **get_interface**, **is_nil, duplicate**, and **release**. (In the CORBA 1.1 specification, these methods are described in Chapter 8.)

In DSOM, there is also another derivation of this class: **SOMDClientProxy**. This subclass inherits the implementation of **SOMDObject**, but extends it by overriding **somDispatch** with a "remote dispatch" method, and caches the binding to the server process. Whenever a remote object is accessed, it is represented in the client process by a **SOMDClientProxy** object.

File Stem

somdobj

Base

SOMObject

Metaclass

SOMClass

Ancestor Classes

SOMObject

New Methods

create_request

create_request_args *

duplicate

get_implementation

get_interface

is_constant *

is_nil

is_proxy *

is_SOM_ref *

release

(* These methods were added by DSOM to supplement the published CORBA 1.1 interfaces.)

Overridden methods

somInit

somUninit

somDumpSelfInt

create_request Method

Purpose

Creates a request to execute a particular operation on the referenced object.

IDL Syntax

ORBStatus create_request (
in Context *ctx*,
in Identifier *operation*,
in NVList *arg_list*,
inout NamedValue *result*,
out Request *request*,
in Flags *req_flags***);**

Description

The **create_request** method creates a request to execute a particular operation on the referenced object. (For more information on the **create_request** call, see CORBA 1.1 page 109.)

In DSOM, this method is meaningful only when invoked on a **SOMDClientProxy** object. If invoked on a **SOMDObject** which is not a client proxy, an exception is returned.

Parameters

receiver	A pointer to a **SOMDObject** object.
env	A pointer to the **Environment** structure for the method caller.
ctx	A pointer to the **Context** object of the requested operation.
operation	The name of the operation to be performed on the target object, *receiver*.
arg_list	A pointer to a list of arguments (**NVList**). If this argument is NULL, the argument list can be assembled by repeated calls to the **add_arg** method on the **Request** object created by calling this method.
result	A pointer to a **NamedValue** structure where the result of applying *operation* to *receiver* should be stored.
request	A pointer to storage for the address of the created **Request** object.
req_flags	A **Flags** bitmask (unsigned long) that may contain the following flag value:

OUT_LIST_MEMORY
: Indicates that any out-arg memory is associated with the argument list. When the list structure is freed, any associated out-arg memory is also freed. If OUT_LIST_MEMORY is specified, an argument list must also have been specified on the **create_request** call.

Return Value

The **create_request** method returns an **ORBStatus** value as the status code for the request.

Example

```
#include <somd.h>
#include <repostry.h>
#include <intfacdf.h>
#include <foo.h> /* provided by user */

/* assume following method declaration in interface Foo:
 *      long methodLong  (in long inLong,inout long inoutLong);
 * then the following code builds a request to execute the call:
 *      result = methodLong(fooObj, &ev, 100,200);
 *using the DII.
 */

Environment ev;
OperationDef opdef;
Description desc;
OperationDescription *opdesc;
NVList arglist;
long rc;
long value1 = 100;
long value2 = 200;
Foo fooObj;
Request reqObj;
NamedValue result;
Identifier name;
TypeCode tc;
void *dummy;
long dummylen;
Flags flags;

/* Get the OperationDef from the Interface Repository. */
opdef = _lookup_id(SOM_InterfaceRepository,
       &ev, ”Foo::methodLong”);
/* Create a NamedValue list for the operation. */
rc= _create_operation_list(SOMD_ORBObject, &ev, opdef, &arglist);

/* Insert arg1 info into arglist */
_get_item(arglist, &ev,
    0, &name, &tc, &dummy, &dummylen, &flags);
_set_item(arglist,&ev,0, name, tc, &value1, sizeof(long), flags);

/* Insert arg2 info into arglist */
_get_item(arglist, &ev,
    1, &name, &tc, &dummy, &dummylen, &flags);
_set_item(arglist,&ev,1, name, tc, &value2, sizeof(long), flags);

/* Get the operation description structure. */
desc = _describe(opdef, &ev);
opdesc = (OperationDescription *) desc.value._value;

/* Fill in the TypeCode field for result. */
result.argument._type = opdesc->result;

/* Finally, create the Request, reqObj */
rc = _create_request(fooObj, &ev, (Context *)NULL, ”methodLong”,
       arglist, &result, &reqObj, (Flags)0);
```

Original Class

SOMDObject

Related Information

Methods: create_request_args, **create_list**, **create_operation_list**

create_request_args Method

Purpose

Creates an argument list appropriate for the specified operation.

IDL Syntax

```
ORBStatus  create_request_args (
                          in Identifier operation,
                          out NVList arg_list.
                          out NamedValue result);
```

Description

The **create_request_args** method creates the appropriate *arg_list* (**NVList**) for the specified operation. It is similar in function to the **create_operation_list** method. Its value is that it also creates the result structure whereas **create_operation_list** does not.

In DSOM, this method is meaningful only when invoked on a **SOMDClientProxy** object. If invoked on a **SOMDObject** which is not a client proxy, an exception is returned.

Parameters

receiver	A pointer to the **SOMDObject** object to create the request.
env	A pointer to the **Environment** structure for the method caller.
operation	The Identifier of the operation for which the argument list is being created.
arg_list	A pointer to the location where the method will store a pointer to the resulting argument list.
result	A pointer to the **NamedValue** structure which will be used to hold the result. The *result*'s type field is filled in with the **TypeCode** of the expected result.

Return Value

The **create_request_args** method returns an **ORBStatus** value representing the return code of the request.

Example

```
#include <somd.h>
#include <repostry.h>
#include <intfacdf.h>
#include <foo.h> /* provided by user */

/* assume following method declaration in interface Foo:
 *      long methodLong  (in long inLong,inout long inoutLong);
 * then the following code builds a request to execute the call:
 *      result = methodLong(fooObj, &ev, 100,200);
 * using the DII.
 */






Environment ev;
OperationDef opdef;
Description desc;
OperationDescription *opdesc;
NVList arglist;
long rc;
long value1 = 100;
long value2 = 200;
Foo fooObj;
Request reqObj;
NamedValue result;
Identifier name;
TypeCode tc;
void *dummy;
long dummylen;
Flags flags;

/* Get the OperationDef from the Interface Repository. */
opdef = _lookup_id(SOM_InterfaceRepository,
       &ev, "Foo::methodLong");
/* Create a NamedValue list for the operation. */
rc= _create_request_args(fooObj, &ev,
         "methodLong", &arglist, &result);

/* Insert arg1 info into arglist */
_get_item(arglist, &ev,
    0, &name, &tc, &dummy, &dummylen, &flags);
_set_item(arglist,&ev,0, name, tc, &value1, sizeof(long), flags);

/* Insert arg2 info into arglist */
_get_item(arglist, &ev,
    1, &name, &tc, &dummy, &dummylen, &flags);
_set_item(arglist,&ev,1, name, tc, &value2, sizeof(long), flags);

/* Finally, create the Request, reqObj */
rc = _create_request(fooObj, &ev, (Context *)NULL, "methodLong",
       arglist, &result, &reqObj, (Flags)0);
```

Original Class

SOMDObject

Related Information

Methods: duplicate, **release**, **create_request**, **create_operation_list**

duplicate Method

Purpose

Makes a duplicate of an object reference.

IDL Syntax

SOMDObject duplicate ( );

Description

The **duplicate** method makes a duplicate of the object reference. The **release** method should be called to free the object.

Parameters

receiver	A pointer to a **SOMDObject** object.
env	A pointer to the **Environment** structure for the method caller.

Return Value

The **duplicate** method returns a **SOMDObject** that is a duplicate of the *receiver. Ownership* of the returned object is transferred to the caller.

Example

```
#include <somd.h>

Environment ev;
SOMObject obj;
SOMDObject objref1, objref2;
...
objref1 = _create_SOM_ref(SOMD_SOMOAObject, &ev, obj);
objref2 = _duplicate(objref1,&ev);
...
_release(objref2,&ev);
```

Original Class

SOMDObject

Related Information

Methods: release, **create**, **create_constant**, **create_SOM_ref**

get_implementation Method

Purpose

Returns the implementation definition for the referenced object.

IDL Syntax

ImplementationDef get_implementation ( );

Description

The **get_implementation** method returns the implementation definition object for the referenced object.

Parameters

receiver	A pointer to a **SOMDObject** object.
env	A pointer to the **Environment** structure for the method caller.

Return Value

The **get_implementation** method returns the **ImplementationDef** object for the *receiver*. *Ownership* of the returned object is transferred to the caller.

Example

```
#include <somd.h>

long flags;
Environment ev;
SOMDObject objref;
ImplementationDef impldef;
...
impldef = _get_implementation(objref,&ev);
flags = __get_impl_flags(impldef,&ev);
```

Original Class

SOMDObject

Related Information

Methods: get_interface

get_interface Method

Purpose

Returns the interface definition object for the referenced object.

IDL Syntax

InterfaceDef get_interface ( );

Description

The **get_interface** method returns the interface definition object for the referenced object.

Parameters

receiver	A pointer to a **SOMDObject** object.
env	A pointer to the **Environment** structure for the method caller.

Return Value

The **get_interface** method returns a pointer to the **InterfaceDef** object associated with the reference *receiver. Ownership* of the **InterfaceDef** object is passed to the caller.

Example

```
#include <somd.h>
#include <repostry.h>
#include <intfacdf.h>

Environment ev;
SOMDObject objref;
InterfaceDef intf;
...
intf = _get_interface(objref,&ev);
```

Original Class

SOMDObject

Related Information

Methods: get_implementation

is_constant Method

Purpose

Tests to see if the object reference is a constant (that is, its **ReferenceData** is a constant value associated with the reference).

IDL Syntax

boolean is_constant ( );

Description

The **is_constant** method tests to see if the object reference was created using the **create_constant** method in the **SOMOA** class.

Parameters

receiver	A pointer to a **SOMDObject** object.
env	A pointer to the **Environment** structure for the method caller.

Return Value

The **is_constant** method returns TRUE if the object reference was generated by the method **create_constant.** Otherwise, **is_constant** returns FALSE.

Example

```
#include <somd.h>

Environment ev;
SOMDObject objref;
...

/* This code might be part of the code
 * that overrides the somdSOMObjFromRef method, i.e.
 * in an implementation of a subclass of SOMDServer called
 * myServer
 */

if (_is_constant(objref, &ev))
  id = _get_id(objref, &ev);

...
```

Related Information

Methods: create, **create_constant**, **is_proxy**, **is_SOM_ref**, **is_nil**

is_nil Method

Purpose

Tests to see if the object reference is nil.

IDL Syntax

boolean is_nil ( );

Description

The **is_nil** method tests to see if the specified object reference is nil.

Parameters

receiver	A pointer to any object, either a **SOMObject** or a **SOMDObject**. The pointer can be NULL.
env	A pointer to the **Environment** structure for the method caller.

Return Value

The **is_nil** method returns TRUE if the object reference is empty. Otherwise, **is_nil** returns FALSE.

Example

```
#include <somd.h>
Environment ev;
SOMDObject objref;
SOMObject somobj;
...
/* This code might be part of the code
 * that overrides the somdSOMObjFromRef method, i.e.
 * in an implementation of a subclass of SOMDServer called
 * myServer
 */
if (_is_nil(objref, &ev) ||
  _somIsA(objref, SOMDClientProxyNewClass(0, 0)) ||
  _is_SOM_ref(objref, &ev)) {
  somobj = myServer_parent_SOMDServer_somdSOMObjFromRef
     (somSelf, &ev, objref);
}
else {
  /* do the myServer-specific stuff to create/find somobj here */
}
return somobj;
```

Related Information

Methods: create, **is_constant**, **is_proxy**, **is_SOM_ref**

is_proxy Method

Purpose

Tests to see if the object reference is a proxy.

IDL Syntax

boolean is_proxy( );

Description

The **is_proxy** method tests to see if the specified object reference is a proxy object.

Parameters

receiver	A pointer to a **SOMDObject** object.
env	A pointer to the **Environment** structure for the method caller.

Return Value

The **is_proxy** method returns TRUE if the object reference is a proxy object. Otherwise, **is_proxy** returns FALSE.

Example

```
#include <somd.h>

SOMDObject objref;
Environment ev;
Context ctx;
NVlist arglist;
NamedValue result;
Request reqObj;
...
if (_is_proxy(objref, &ev)) {
  /* create a remote request for target object */
...
 rc = _create_request(obj, &ev, ctx,
        ”testMethod”, arglist, &result, &reqObj,
        (Flags)0);
}
...
```

Original Class

SOMDObject

Related Information

Methods: is_nil, **is_constant**, **is_SOM_ref**, **string_to_object**

is_SOM_ref Method

Purpose

Tests to see if the object reference is a simple reference to a SOM object.

IDL Syntax

boolean is_SOM_ref ( );

Description

The **is_SOM_ref** method tests to see if the specified object reference is a simple (transient) reference to a SOM object.

Parameters

receiver	A pointer to a **SOMDObject** object.
env	A pointer to the **Environment** structure for the method caller.

Return Value

The **is_SOM_ref** method returns TRUE if the object reference is a simple (transient) reference to a SOM object. Otherwise, **is_SOM_ref** returns FALSE.

Example

```
#include <somd.h>

SOMDObject objref;
Environment ev;
SOMObject obj;
...
if (_is_SOM_ref(objref, &ev))
  /* we know objref is a simple reference, so we can ... */
  obj = _get_SOM_object(SOMD_SOMOAObject, &ev, objref);
...
```

Original Class

SOMDObject

Related Information

Methods: create_SOM_ref, **get_SOM_object**, **is_proxy**, **is_nil**, **is_constant**

release Method

Purpose

Releases the memory associated with the specified object reference.

IDL Syntax

void release ( );

Description

The **release** method releases the memory associated with the object reference.

Parameters

receiver	A pointer to a **SOMDObject** object.
env	A pointer to the **Environment** structure for the method caller.

Example

```
#include <somd.h>

SOMDObject objref;
Environment ev;
SOMObject obj;
...
objref = _create_SOM_ref(SOMD_SOMOAObject, &ev, obj);
...
_release(objref, &ev);
```

Original Class

SOMDObject

Related Information

Methods: duplicate, **somdReleaseObject**, **somdProxyFree**, **create**, **create_constant**, **create_SOM_ref**, **somdReleaseResources**

SOMDObjectMgr Class

Description

The **SOMDObjectMgr** class is derived from **ObjectMgr** class and provides the DSOM implementations for the **ObjectMgr** methods.

File Stem

somdom

Base

ObjectMgr

Metaclass

SOMMSingleInstance

Ancestor Classes

ObjectMgr

SOMObject

Attribute

Listed below is an available **SOMDObjectMgr** attribute, with its corresponding type in parentheses, followed by a description of its purpose:

somd21somFree (boolean)

Determines whether or not **somFree**, when invoked on a proxy object, will free the proxy object along with the remote object. The default value is FALSE, indicating that only the remote object will be freed when **somFree** is invoked on a proxy object. Setting this attribute to TRUE as part of client-program initialization, for example,

```
__set_somd21somdFree(SOMD_ObjectMgr, ev, TRUE);
```

has the effect that all subsequent invocations of **somFree** on proxy objects will free both the remote object and the proxy.

New Methods

somdFindAnyServerByClass *

somdFindServer *

somdFindServerByName *

somdFindServersByClass *

(* This class and its methods were added by DSOM to supplement the published CORBA 1.1 interfaces.)

Overridden Methods

somdDestroyObject

somdGetIdFromObject

somdGetObjectFromId

somdNewObject

somdReleaseObject

somInit

somdFindAnyServerByClass Method

Purpose

Finds a server capable of creating the specified object.

IDL Syntax

SOMDServer somdFindAnyServerByClass (
in Identifier *objclass***);**

Description

The **somdFindAnyServerByClass** method finds a server capable of creating an object of the specified type with the specified properties.

Parameters

receiver	A pointer to a **SOMDObjectMgr** object.
env	A pointer to the **Environment** structure for the method caller.
objclass	An **Identifier** specifying the class of the object the server needs to be able to create.

Return Value

The **somdFindAnyServerByClass** method returns a pointer to a **SOMDServer** proxy. Or, if no server can be found in the Implementation Repository that implements the specified class, NULL is returned.

Example

```
#include <somd.h>
#include <stack.h> /* provided by user */

Stack stk;
Environment ev;
SOMDServer server;

SOM_InitEnvironment(&ev);
SOMD_Init(&ev);
StackNewClass(0,0);
server =
  _somdFindAnyServerByClass(SOMD_ObjectMgr, &ev, ”Stack”);
stk = _somdCreateObj(server, &ev, ”Stack”, ””);
...
_somdDestroyObject(SOMD_ObjectMgr, &ev, stk);
```

Original Class

SOMDObjectMgr

Related Information

Methods: somdFindServer, **somdFindServerByName**, **somdFindServersByClass**

somdFindServer Method

Purpose

Finds a server given its **ImplementationDef** ID.

IDL Syntax

SOMDServer somdFindServer (
in ImplId *serverid***);**

Description

The **somdFindServer** method finds a server capable of creating an object of the specified type with the specified properties.

Parameters

receiver	A pointer to a **SOMDObjectMgr** object.
env	A pointer to the **Environment** structure for the method caller.
serverid	An **ImplId** string which identifies the **ImplementationDef** of the desired server.

Return Value

The **somdFindServer** method returns a pointer to a **SOMDServer** proxy.

Example

```
#include <somd.h>
#include <stack.h> /* provided by user */

Stack stk;
Environment ev;
SOMDServer server;
ImplId implid;

SOM_InitEnvironment(&ev);
SOMD_Init(&ev);
StackNewClass(0,0);
server = _somdFindServer(SOMD_ObjectMgr, &ev, implid);
stk = _somdCreateObj(server, &ev, ”Stack”, ””);
...
_somdDestroyObject(SOMD_ObjectMgr, &ev, stk);
```

Original Class

SOMDObjectMgr

Related Information

Methods:, **somdFindServerByName**, **somdFindServersByClass**, **somdFindAnyServerByClass**

somdFindServerByName Method

Purpose

Finds a server given its **ImplementationDef** name (alias).

IDL Syntax

SOMDServer somdFindServerByName (
in string *servername***);**

Description

The **somdFindServerByName** method finds a server with the specified name.

Parameters

receiver	A pointer to a **SOMDObjectMgr** object.
env	A pointer to the **Environment** structure for the method caller.
servername	An **string** which specifies the name of the **ImplementationDef** of the desired server.

Return Value

The **somdFindServerByName** method returns a pointer to a **SOMDServer** proxy.

Example

```
#include <somd.h>
#include <stack.h> /* provided by user */

Stack stk;
Environment ev;
SOMDServer server;

SOM_InitEnvironment(&ev);
SOMD_Init(&ev);
StackNewClass(0,0);
server =
   _somdFindServerByName(SOMD_ObjectMgr, &ev, ”stackServer”);
stk = _somdCreateObj(server, &ev, ”Stack”, ””);
...
_somdDestroyObject(SOMD_ObjectMgr, &ev, stk);
```

Original Class

SOMDObjectMgr

Related Information

Methods: somdFindServer, **somdFindServersByClass**, **somdFindAnyServerByClass**

somdFindServersByClass Method

Purpose

Finds all servers capable of creating a particular object.

IDL Syntax

sequence<SOMDServer> somdFindServersByClass (
in Identifier *objclass***);**

Description

The **somdFindServersByClass** method finds all servers capable of creating a particular object with the specified properties.

Parameters

receiver	A pointer to a **SOMDObjectMgr** object.
env	A pointer to the **Environment** structure for the method caller.
objclass	An **Identifier** representing the type of the object the server needs to be able to create.

Return Value

The **somdFindServersByClass** method returns a sequence of **SOMDServer** objects capable of creating the specified object.

Example

```
#include <somd.h>
#include <stack.h> /* provided by user */

Stack stk;
Environment ev;
sequence(SOMDServer) servers;
SOMDServer server;
SOMDServer chooseServer(sequence(SOMDServer) servers);

SOM_InitEnvironment(&ev);
SOMD_Init(&ev);
StackNewClass(0,0);
servers = _somdFindServersByClass(SOMD_ObjectMgr, &ev, ”Stack”);
server = chooseServer(servers);
stk = _somdCreateObj(server, &ev, ”Stack”, ””);
...
_somdDestroyObject(SOMD_ObjectMgr, &ev, stk);
```

Original Class

SOMDObjectMgr

Related Information

Methods: somdFindServer, **somdFindServerByName**, **somdFindAnyServerByClass**

SOMDServer Class

Description

The **SOMDServer** class is a base class that defines and implements methods for managing objects in a DSOM server process. This includes methods for the creation and deletion of SOM objects, and for getting the SOM class object for a specified class. The **SOMDServer** class also defines and implements methods for the mapping between object references (**SOMDObject**s) and SOM objects, and dispatching methods on objects.

Application-specific methods for managing application objects can be introduced in subclasses of **SOMDServer.**

File Stem

SOMDServer

Base

SOMObject

Metaclass

SOMMSingleInstance

Ancestor Classes

SOMObject

New Methods

somdCreateObj *

somdDeleteObj *

somdDispatchMethod *

somdGetClassObj *

somdObjReferencesCached *

somdRefFromSOMObj *

somdSOMObjFromRef *

(* This class and its methods were added by DSOM to supplement the published CORBA 1.1 interfaces.)

somdCreateObj Method

Purpose

Creates an object of the specified class.

IDL Syntax

SOMObject somdCreateObj (
in Identifier *objclass*,
in string *hints***);**

Description

The **somdCreateObj** method creates an object of the specified class.

Parameters

receiver	A pointer to a **SOMDServer** object capable of creating an instance of the specified class.
env	A pointer to the **Environment** structure for the method caller.
objclass	The class of the object for which an instance is to be created.
hints	A **string** which may optionally be used to specify special creation options.

Return Value

The **somdCreateObj** method returns a **SOMObject** of the class specified by *objclass*.

Example

```
#include <somd.h>
#include <stack.h> /* provided by user */

Stack stk;
Environment ev;
SOMDServer server;

SOM_InitEnvironment(&ev);
SOMD_Init(&ev);
StackNewClass(0,0);
server =
  _somdFindServerByName(SOMD_ObjectMgr, &ev,"stackServer");
stk = _somdCreateObj(server, &ev, "Stack", "");
...
_somdDestroyObject(SOMD_ObjectMgr, &ev, stk);
```

Original Class

SOMDServer

somdDeleteObj Method

Purpose

Deletes the specified object.

IDL Syntax

void somdDeleteObj (
in SOMObject *somobj* **);**

Description

The **somdDeleteObj** method deletes the specified object.

Parameters

receiver	A pointer to a **SOMDServer** object.
env	A pointer to the **Environment** structure for the method caller.
somobj	An object “managed” by the server object.

Example

```
#include <somd.h>
#include <stack.h> /* provided by user */

Stack stk;
Environment ev;
SOMDServer server;

SOM_InitEnvironment(&ev);
SOMD_Init(&ev);
StackNewClass(0,0);
server =
  _somdFindServerByName(SOMD_ObjectMgr, &ev,”stackServer”);
stk = _somdCreateObj(server, &ev, ”Stack”, ””);
...
_somdDeleteObj(server, &ev, stk);
```

Original Class

SOMDServer

somdDispatchMethod Method

Purpose

Dispatch a method on the specified SOM object.

IDL Syntax

void somdDispatchMethod (
in SOMObject *somobj*,
out somToken *retValue*,
in somId *methodId*,
in va_list *ap* **);**

Description

The **somdDispatchMethod** method is used to intercept method calls on objects in a server. When a request arrives, the request parameters are extracted from the message, and the target object is resolved. Then, the **SOMOA** dispatches the method call on the target object using the **somdDispatchMethod** method.

The default implementation will call **somDispatch** on the target object with the parameters as specified. This method can be overridden to intercept and process the method calls before they are dispatched.

Parameters

receiver	A pointer to a **SOMDServer** object.
env	A pointer to the **Environment** structure for the method caller.
somobj	A pointer to an object “managed” by the server object.
retValue	A pointer to the storage area allocated to hold the method result value, if any.
methodId	A **somId** for the name of the method which is to be dispatched.
ap	A pointer to a **va_list** array of arguments to the method call.

Return Value

The **somdDispatchMethod** method will return a result, if any, in the storage whose address is in *retValue.*

Example

```
#include <somd.h>

/* overridden somdDispatchMethod */
void somdDispatchMethod(SOMDServer *somself, Environment *ev,
       SOMObject *somobj, somToken *retValue,
       somId methodId, va_list ap)
{
 printf(”dispatching %s on %x\n”, SOM_StringFromId(methodId),
somobj);
 SOMObject_somDispatch(somobj, ev, retValue, methodId, ap);
}
```

Original Class

SOMDServer

somdGetClassObj Method

Purpose

Creates a class object for the specified class.

IDL Syntax

SOMClass somdGetClassObj (
in Identifier *objclass***);**

Description

The **somdGetClassObj** method creates a class object of the specified type.

Parameters

receiver	A pointer to a **SOMDServer** object.
env	A pointer to the **Environment** structure for the method caller.
objclass	An identifier specifying the type of the class object to be created.

Return Value

The **somdGetClassObj** method returns a **SOMClass** object of the type specified.

Example

```
#include <somd.h>
#include <stack.h> /* provided by user */

SOMClass stkclass;
Environment ev;
SOMDServer server;

SOM_InitEnvironment(&ev);
SOMD_Init(&ev);
StackNewClass(0,0);
server =
  _somdFindServerByName(SOMD_ObjectMgr, &ev,"stackServer");
stkclass = _somdGetClassObj(server, &ev, "Stack", "");
```

Original Class

SOMDServer

somdObjReferencesCached Method

Purpose

Indicates whether a server object retains ownership of the object references it creates via the **somdRefFromSOMObj** method.

Syntax

boolean somdObjReferencesCached ( );

Description

The **somdObjReferencesCached** method indicates whether a server object retains ownership of the object references it creates via the **somdRefFromSOMObj** method. The default implementation returns FALSE, meaning that the server turns over ownership of the object references it creates to the caller. Subclasses of **SOMDServer** that implement object reference caching should override this method to return TRUE.

Parameters

receiver	A pointer to an object of class **SOMDServer.**
ev	A pointer to the **Environment** structure for the calling method.

Return Value

The method returns FALSE by default; overriding implementations may return TRUE to indicate that a subclass of **SOMDServer** implements object reference caching.

Example

```
SOMDobject objref;
objref = _somdRefFromSOMObj(serverObj, ev, myobj);
...
/* code to use objref */
...
if (!_somdObjReferencesCached(serverObj, ev))
  _release(objref, ev);
```

Original Class

SOMDServer

Related Information

Methods: somdRefFromSOMObj

somdRefFromSOMObj Method

Purpose

Returns an object reference corresponding to the specified SOM object.

IDL Syntax

SOMDObject somdRefFromSOMObj (
in SOMObject *somobj***);**

Description

The **somdRefFromSOMObj** method creates a simple (transient) reference to a SOM object. This method is called by **SOMOA** as part of converting the results of a local method call into a result message for a remote client.

By default the **somdRefFromSOMObj** method turns over ownership of the object reference it creates to the caller. However, if a subclass of **SOMDServer** overrides **somdRefFromSOMObj** to implement object reference caching, then that subclass should also override the method **somdObjReferencesCached** to report that caching by returning TRUE.

Parameters

receiver	A pointer to a **SOMDServer** object.
env	A pointer to the **Environment** structure for the method caller.
somobj	A pointer to the SOM object for which a DSOM reference is to be created.

Return Value

The **somdRefFromSOMObj** method returns a DSOM reference (that is, a **SOMDObject**) for the SOM object specified.

Example

```
#include <somd.h>
#include <stack.ih> /* user-generated */

SOMDObject objref;
Environment ev;
SOMObject obj;
...
/* myServer specific code up here */
...
/* one might want to make this call as part of the code
 * that overrides the somdRefFromSOMObj method, i.e.
 * in an implementation of a subclass of SOMDServer called
 * myServer
 */
objref =
 myServer_parent_SOMDServer_somdRefFromSOMObj(somSelf, &ev, obj);
```

Original Class

SOMDServer

Related Information

Method: somdObjReferencesCached

somdSOMObjFromRef Method

Purpose

Returns the SOM object corresponding to the specified object reference.

IDL Syntax

SOMObject somdSOMObjFromRef (
in SOMDObject *objref***);**

Description

The **somdSOMObjFromRef** method returns the SOM object associated with the DSOM object reference, *objref*. This method is called by **SOMOA** as part of converting a remote request into a local method call on an object.

Parameters

receiver	A pointer to a **SOMDServer** object.
env	A pointer to the **Environment** structure for the method caller.
objref	A pointer to the DSOM object reference to the SOM object.

Return Value

The **somdSOMObjFromRef** method returns the SOM object associated with the supplied DSOM reference.

Example

```
#include <somd.h>
#include <stack.ih> /* user-generated */

SOMDObject objref;
Environment ev;
SOMObject obj;
...
/* myServer specific code up here */
...
/* one might want to make this call as part of the code
 * that overrides the somdRefFromSOMObj method, i.e.
 * in an implementation of a subclass of SOMDServer called
 * myServer
 */
obj =
myServer_parent_SOMDServer_somdSOMObjFromRef(somSelf,&ev,objref);
```

Original Class

SOMDServer

SOMDServerMgr Class

Description

The **SOMDServerMgr** class provides a programmatic interface to manage server processes. At present, the server processes that can be managed are limited to those present in the Implementation Repository. The choice of Implementation Repository is determined by the environment variable SOMDDIR.

File Stem

servmgr

Base

SOMObject

Metaclass

SOMClass

Ancestor Classes

SOMObject

New Methods

somdDisableServer

somdEnableServer

somdIsServerEnabled

somdListServer

somdRestartServer

somdShutdownServer

somdStartServer

somdDisableServer Method

Purpose

Disables a server process from starting until it is explicitly enabled again.

IDL Syntax

ORBStatus somdDisableServer (in string *server_alias***);**

Description

The **somdDisableServer** method disables the server process associated with the server alias. Once a server process has been disabled, it cannot be restarted until it is explicitly enabled again. Initially, all server processes are enabled by default. Note: If the server process to be disabled is currently running, then it is first stopped before disabling. If the method is unsuccessful in stopping the server, the disable method fails.

Parameters

receiver	A pointer to an object of class **SOMDServerMgr**.
ev	A pointer to the **Environment** structure for the calling method.
server_alias	The implementation alias of the server to be disabled.

Return Value

Returns 0 for success or a DSOM error code for failure.

Example

```
#include <somd.h>
#include <servmgr.h>

SOMDServerMgr servmgr;
string  server_alias = ”MyServer”;
ORBStatus rc;
Environment e;

SOM_InitEnvironment(&e);
SOMD_Init(&e);
servmgr = SOMDServerMgrNew();
rc = _somdDisableServer(servmgr, &e, server_alias);
```

Original Class

SOMDServerMgr

Related Information

Methods: somdEnableServer

somdEnableServer Method

Purpose

Enables a server process so that it can be started when required. Initially, all server processes are enabled by default.

IDL Syntax

ORBStatus somdEnableServer (in string *server_alias***);**

Description

The **somdEnableServer** method enables a server process associated with the server alias. Initially, all server processes are enabled by default. Server processes can be disabled by using the **somdDisableServer** method.

Parameters

receiver	A pointer to an object of class **SOMDServerMgr**.
ev	A pointer to the **Environment** structure for the calling method.
server_alias	The implementation alias of the server to be enabled.

Return Value

Returns 0 for success or a DSOM error code for failure.

Example

```
SOMDServerMgr servmgr;
string  server_alias = ”MyServer”;
ORBStatus rc;
Environment e;

SOM_InitEnvironment(&e);
SOMD_Init(&e);
servmgr = SOMDServerMgrNew();

/* disable the server */
rc = _somdDisableServer(servmgr, &e, server_alias);

/*  do some processing */

/* enable the server */
rc = _somdEnableServer(servmgr, &e, server_alias);
```

Original Class

SOMDServerMgr

Related Information

Methods: somdDisableServer

somdIsServerEnabled Method

Purpose

Determines whether a server process is enabled or not.

IDL Syntax

boolean somdIsServerEnabled (in ImplementationDef *impldef***);**

Description

The **somdIsServerEnabled** method returns a **boolean** corresponding to the current state (enabled/disabled) of the server process.

Parameters

receiver	A pointer to an object of class **SOMDServerMg**r.
ev	A pointer to the **Environment** structure for the calling method.
impldef	A pointer to the **ImplementationDef** object for the server, obtained using the **find_impldef_by_alias** method when it is invoked on the global **SOMD_ImplRepObject**.

Return Value

Returns TRUE if the server is enabled; otherwise, FALSE is returned.

Example

```
#include <somd.h>
#include <servmgr.h>

SOMDServerMgr servmgr;
ImplementationDef impldef;
string  server_alias = ”MyServer”;
boolean rc;
Environment e;

SOM_InitEnvironment(&e);
SOMD_Init(&e);

impldef = _find_impldef_by_alias(SOMD_ImplRepObject,
             &e, server_alias);
servmgr = SOMDServerMgrNew();

/* if server is disabled then enable it*/
if (!_somdIsServerEnabled(servmgr, &e, impldef))
  rc = _somdEnableServer(servmgr, &e, server_alias);
```

Original Class

SOMDServerMgr

Related Information

Methods: somdDisableServer, **somdEnableServer**

somdListServer Method

Purpose

Queries the state of a server process.

IDL Syntax

ORBStatus somdListServer (in string *server_alias***);**

Description

The **somdListServer** method is invoked to query the status of the server process associated with the server alias. If the server process is running, the return code will be 0 indicating success. Status codes of SOMDERROR_ServerDisabled or SOMDERROR_ServerNotFound may also be returned. The former return code indicates that the server process has been disabled (refer **somdDisableServer**) and the latter indicates that the server process is not currently running.

Parameters

receiver	A pointer to an object of class **SOMDServerMgr**.
ev	A pointer to the **Environment** structure for the calling method.
server_alias	The implementation alias of the server to be listed.

Return Value

Returns 0 if the server process is running; otherwise, a DSOM error code is returned.

Example

```
#include <somd.h>
#include <servmgr.h>

SOMDServerMgr servmgr;
string  server_alias = ”MyServer”;
ORBStatus rc;
Environment e;

SOM_InitEnvironment(&e);
SOMD_Init(&e);
servmgr = SOMDServerMgrNew();
rc = _somdListServer(servmgr, &e, server_alias);
if (!rc)        /* server is running */
  rc = _somdShutdownServer(servmgr, &e, server_alias);
else if (rc == SOMDERROR_ServerNotFound)
          /* server is not running */
  rc = _somdStartServer(servmgr, &e, server_alias);
```

Original Class

SOMDServerMgr

somdRestartServer Method

Purpose

Restarts a server process.

IDL Syntax

ORBStatus somdRestartServer (in string *server_alias***);**

Description

The **somdRestartServer** method is invoked to restart a server process. If the server process currently exists, it will be stopped and started again. If the server process does not exist, a new server process will still be started. If the server process cannot be stopped and/or started for any reason, the method returns a DSOM error code.

Parameters

receiver	A pointer to an object of class **SOMDServerMgr**.
ev	A pointer to the **Environment** structure for the calling method.
server_alias	The implementation alias of the server to be restarted.

Return Value

Returns 0 for success or a DSOM error code for failure.

Example

```
#include <somd.h>
#include <servmgr.h>

SOMDServerMgr servmgr;
string  server_alias = ”MyServer”;
ORBStatus rc;
Environment e;

SOM_InitEnvironment(&e);
SOMD_Init(&e);
servmgr = SOMDServerMgrNew();
rc = _somdRestartServer(servmgr, &e, server_alias);
```

Original Class

SOMDServerMgr

somdShutdownServer Method

Purpose

Stops a server process.

IDL Syntax

ORBStatus somdShutdownServer (in string *server_alias***);**

Description

The **somdShutdownServer** method is invoked to stop a server process. If the server process corresponding to the server alias exists, it will be stopped and a code indicating success is returned. If the server process does not exist, then the SOMDERROR_ServerNotFound error is returned.

Note: On AIX, this method will fail to stop the server process if the process owner executing this method is not the same as that of either the server process *or* root.

Parameters

receiver	A pointer to an object of class **SOMDServerMgr**.
ev	A pointer to the **Environment** structure for the calling method.
server_alias	The implementation alias of the server to be stopped.

Return Value

Returns 0 for success or a DSOM error code for failure.

Example

```
#include <somd.h>
#include <servmgr.h>

SOMDServerMgr servmgr;
string  server_alias = ”MyServer”;
ORBStatus rc;
Environment e;

SOM_InitEnvironment(&e);
SOMD_Init(&e);
servmgr = SOMDServerMgrNew();
rc = _somdShutdownServer(servmgr, &e, server_alias);
```

Original Class

SOMDServerMgr

somdStartServer Method

Purpose

Starts a server process.

IDL Syntax

ORBStatus somdStartServer (in string *server_alias***);**

Description

The **somdStartServer** method is invoked to start a server process. If the server process does not exist, the server process is started and the code indicating success is returned. If the server process already exists, then the return code will still indicate success and the server process will be undisturbed.

Parameters

receiver	A pointer to an object of class **SOMDServerMgr**.
ev	A pointer to the **Environment** structure for the calling method.
server_alias	The implementation alias of the server to be started.

Return Value

Returns 0 for success or a DSOM error code for failure.

Example

```
#include <somd.h>
#include <servmgr.h>

SOMDServerMgr servmgr;
string  server_alias = ”MyServer”;
ORBStatus rc;
Environment e;

SOM_InitEnvironment(&e);
SOMD_Init(&e);
servmgr = SOMDServerMgrNew();
rc = _somdStartServer(servmgr, &e, server_alias);
```

Original Class

SOMDServerMgr

SOMOA Class

Description

The **SOMOA** class is DSOM's basic object adapter. **SOMOA** is a subclass of the abstract **BOA** class, and provides implementations of all the **BOA** methods. The **SOMOA** class also introduces methods for receiving and dispatching requests on SOM objects. **SOMOA** provides some additional methods for creating and managing object references.

File Stem

somoa

Base

BOA

Metaclass

SOMMSingleInstance

Ancestor Classes

BOA **SOMObject**

New Methods

activate_impl_failed *

change_id *

create_constant *

create_SOM_ref *

execute_next_request *

execute_request_loop *

get_SOM_object *

(* This class and its methods were added by DSOM to supplement the published CORBA 1.1 interfaces.)

Overridden Methods

change_implementation

create

deactivate_impl

deactivate_obj

dispose

get_id

get_principal

impl_is_ready

obj_is_ready

set_exception

activate_impl_failed Method

Purpose

Sends a message to the DSOM daemon indicating that a server did not activate.

IDL Syntax

void activate_impl_failed (
in ImplementationDef *implDef*,
in long *rc*);

Description

The **activate_impl_failed** method sends a message to the DSOM daemon (**somdd**) indicating that the server did not activate.

Parameters

receiver	A pointer to the **SOMOA** object that attempted to activate the implementation.
env	A pointer to the **Environment** structure for the method caller.
implDef	A pointer to the **ImplementationDef** object representing the implementation that failed to activate.
rc	A return code designating the reason for failure.

Example

```
#include <somd.h> /* needed by all servers */
main(int argc, char **argv)
{
 Environment ev;
 SOM_InitEnvironment(&ev);

 /* Initialize the DSOM run-time environment */
 SOMD_Init(&ev);

 /* Retrieve its ImplementationDef from the Implementation
  Repository by passing its implementation ID as a key */
 SOMD_ImplDefObject =
  _find_impldef(SOMD_ImplRepObject, &ev, argv[1]);

 /* create the SOMOA */
 SOMD_SOMOAObject = SOMOANew();
...
/* suppose something went wrong with server initialization */
...
/* tell the daemon (via SOMOA) that activation failed */
_activate_impl_failed(SOMD_SOMOAObject,
        &ev, SOMD_ImplDefObject, rc);
```

Original Class

SOMOA

change_id Method

Purpose

Changes the reference data associated with an object.

IDL Syntax

void change_id (
in SOMDObject *objref*,
in ReferenceData *id* **);**

Description

The **change_id** changes the **ReferenceData** associated with the object identified by *objref.* The **ReferenceData** previously stored in the **SOMOA**'s reference data table is replaced with the value of *id*. The new ID cannot be larger than the maximum size of the original **ReferenceData** (usually specified as 1024 bytes).

Parameters

receiver	A pointer to the **SOMOA** object managing the implementation.
env	A pointer to the **Environment** structure for the method caller.
objref	A pointer to the **SOMDObject** which identifies the object.
id	A pointer to the **ReferenceData** structure representing the object to be created.

Example

```
#include <somd.h>
#include <repostry.h>
#include <intfacdf.h>

Environment ev;
ReferenceData id;
InterfaceDef intfdef;
SOMDObject objref;
string fname; /* file name to be saved with reference */
...
/* create the id for the reference */
id._maximum = id._length = strlen(fname)+1;
id._buffer = (string) SOMMalloc(strlen(fname)+1);
strcpy(id._buffer,fname);

/* get the interface def object for interface Foo*/
intfdef = _lookup_id(SOM_InterfaceRepository, &ev, ”Foo”);

objref = _create_constant(SOMD_SOMOAObject,
       &ev, id, intfdef, SOMD_ImplDefObject);
```

create_constant Method

Purpose

Creates a “constant” object reference.

IDL Syntax

```
SOMDObject  create_constant (
                                in ReferenceData id,
                                in InterfaceDef intf,
                                in ImplementationDef impl);
```

Description

The **create_constant** method is a variant of the **create** method. Like **create**, it creates an object reference for an object with the specified interface and associates the supplied **ReferenceData** with the object reference. The **ReferenceData** can later be retrieved using the **get_id** method. Unlike **create**, this method creates a “constant” reference whose ID value cannot be changed. (See the **change_id** Method on page 2-131.) This is because the ID is maintained as a constant part of the object reference state, versus stored in the reference data table for the server.

This method would be used whenever the application prefers not to maintain an object’s **ReferenceData** in the server’s reference data table.

Parameters

receiver	A pointer to the **SOMOA** object managing the implementation.
env	A pointer to the **Environment** structure for the method caller.
id	A pointer to the **ReferenceData** structure containing application-specific information describing the target object.
intf	A pointer to the **InterfaceDef** object which describes the interface of the target object.
impl	A pointer to the **ImplementationDef** object which describes the application (server) process which implements the target object.

Return Value

The **create_constant** method returns a pointer to a **SOMDObject**. *Ownership* of the new object reference is transferred to the caller.

Example

```
#include <somd.h>
#include <repostry.h>
#include <intfacdf.h>

Environment ev;
ReferenceData id;
InterfaceDef intfdef;
SOMDObject objref;
string fname; /* file name to be saved with reference */
...
/* create the id for the reference */
id._maximum = id._length = strlen(fname)+1;
id._buffer = (string) SOMMalloc(strlen(fname)+1);
strcpy(id._buffer,fname);

/* get the interface def object for interface Foo*/
intfdef = _lookup_id(SOM_InterfaceRepository, &ev, ”Foo”);

objref = _create_constant(SOMD_SOMOAObject,
       &ev, id, intfdef, SOMD_ImplDefObject);
```

Original Class

SOMOA

Related Information

Methods: create, **create_SOM_ref**, **dispose**, **get_id**, **is_constant**

create_SOM_ref Method

Purpose

Creates a simple, transient DSOM reference to a SOM object.

IDL Syntax

SOMDObject create_SOM_ref (
in SOMObject *somobj*,
in ImplementationDef *impl***);**

Description

The **create_SOM_ref** method creates a simple DSOM reference (**SOMDObject**) for a local SOM object. The reference is "special" in that there is no explicit **ReferenceData** associated with the object. Also, this object reference is only valid while the target SOM object exists.

The **SOMObject** associated with the SOM_ref can be retrieved via the **get_SOM_object** method. The **is_SOM_ref** method of **SOMDObject** can be used to determine whether the reference was created using **create_SOM_ref** or not.

Parameters

receiver	A pointer to the **SOMOA** object managing the implementation.
env	A pointer to the **Environment** structure for the method caller.
somobj	A pointer to the local **SOMObject** to be referenced.
impl	A pointer to the **ImplementationDef** of the calling server process.

Return Value

The **create_SOM_ref** method returns a pointer to a **SOMDObject**. *Ownership* of the new object reference is transferred to the caller.

Example

```
#include <somd.h>

SOMDObject objref;
Environment ev;
SOMObject obj;
...
/* one might want to make this call as part of the code
 * that overrides the somdRefFromSOMObj method, i.e.
 * in an implementation of a subclass of SOMDServer.
 */
objref = _create_SOM_ref(SOMD_SOMOAObject, &ev, obj);
```

Original Class

SOMOA

Related Information

Methods: get_SOM_object, **is_SOM_ref**

execute_next_request Method

Purpose

Receive a request message, execute the request, and return to the caller.

IDL Syntax

ORBStatus execute_next_request (
in Flags *waitFlag* **);**

Description

The **execute_next_request** method receives the next request message, executes the request, and sends the result to the caller.

If the server's **ImplementationDef** indicates the server is multi-threaded (the **impl_flags** has the IMPLDEF_MULTI_THREAD flag set), each request will be run by **SOMOA** in a separate thread.

Parameters

receiver	A pointer to the **SOMOA** object managing the implementation.
env	A pointer to the **Environment** structure for the method caller.
waitFlag	A **Flags** value (unsigned long) indicating whether the method should block if there is no message pending (SOMD_WAIT) or return with an error (SOMD_NO_WAIT).

Return Value

The **execute_next_request** method returns an **ORBStatus** value representing the return value for the operation. SOMDERROR_NoMessages is returned if the method is invoked with SOMD_NO_WAIT and no message is available.

Example

```
#include <somd.h>

/* server initialization code ... */
SOM_InitEnvironment(&ev);

/* signal DSOM that server is ready */
_impl_is_ready(SOMD_SOMOAObject, &ev, SOMD_ImplDefObject);

while (ev._major == NO_EXCEPTION) {
  (void) _execute_next_request(SOMD_SOMOAObject,&ev,SOMD_WAIT);
  /* perform appl-specific code between messages here, e.g.,*/
  numMessagesProcessed++;
}
```

Original Class

SOMOA

Related Information

Methods: execute_request_loop

execute_request_loop Method

Purpose

Receives a request message, executes the request, and returns the result to the calling client.

IDL Syntax

ORBStatus execute_request_loop (
in Flags *waitFlag***);**

Description

The **execute_request_loop** method initiates a loop that waits for a request message, executes the request, and returns the result to the client who invoked the request. When called with the SOMD_WAIT flag, this method loops infinitely (or until an error). When called with the SOMD_NO_WAIT flag, this method loops as long as it finds a request message to process.

The SOMD_NO_WAIT flag is useful when writing event-driven applications where there are event sources other than DSOM requests (for example, user input). In this case, DSOM cannot be given exclusive control. Instead, a DSOM event handler can be written using the SOMD_NO_WAIT option, to process all pending requests before returning control to the event manager.

If the server's **ImplementationDef** indicates the server is multi-threaded (the **impl_flags** has the IMPLDEF_MULTI_THREAD flag set), each request will be run by **SOMOA** in a separate thread (OS/2 only).

Parameters

receiver	A pointer to the **SOMOA** object managing the implementation.
env	A pointer to the **Environment** structure for the method caller.
waitFlag	A **Flags** bitmask (unsigned long) indicating whether the method should block (SOMD_WAIT) or return to the caller (SOMD_NO_WAIT) when there is no request message pending.

Return Value

The **execute_request_loop** method may return an **OBJ_ADAPTER** exception which contains an DSOM error code for the operation. SOMDERROR_NoMessages is returned as an **ORBStatus** code if the method is invoked with SOMD_NO_WAIT and no message is pending.

Example

```
#include <somd.h>

/* server initialization code ... */
...
_impl_is_ready(SOMD_SOMOAObject, &ev, SOMD_ImplDefObject);

/* turn control over to SOMOA */
(void) _execute_request_loop(SOMD_SOMOAObject, &ev, SOMD_WAIT);
```

Original Class

SOMOA

Related Information

Functions: SOMD_RegisterCallback

Methods: execute_next_request

See Chapter 12 of the *SOM Toolkit User's Guide* for a description of the Event Management (EMan) framework, for writing event-driven applications.

get_SOM_object Method

Purpose

Get the SOM object associated with a simple DSOM reference.

IDL Syntax

SOMObject get_SOM_object (
in SOMDObject *somref***);**

Description

The **get_SOM_object** method returns the SOM object associated with a reference created by the **create_SOM_ref** method.

Parameters

receiver	A pointer to the **SOMOA** object managing the implementation.
env	A pointer to the **Environment** structure for the method caller.
somref	A pointer to a **SOMDObject** created by the **create_SOM_ref** method.

Return Value

The **get_SOM_object** method returns the SOM object associated with the reference.

Example

```
#include <somd.h>

SOMDObject objref;
Environment ev;
SOMObject obj;
...
if (_is_SOM_ref(objref, &ev))
  /* we know objref is a simple reference, so we can ... */
  obj = _get_SOM_object(SOMD_SOMOAObject, &ev, objref);
...
```

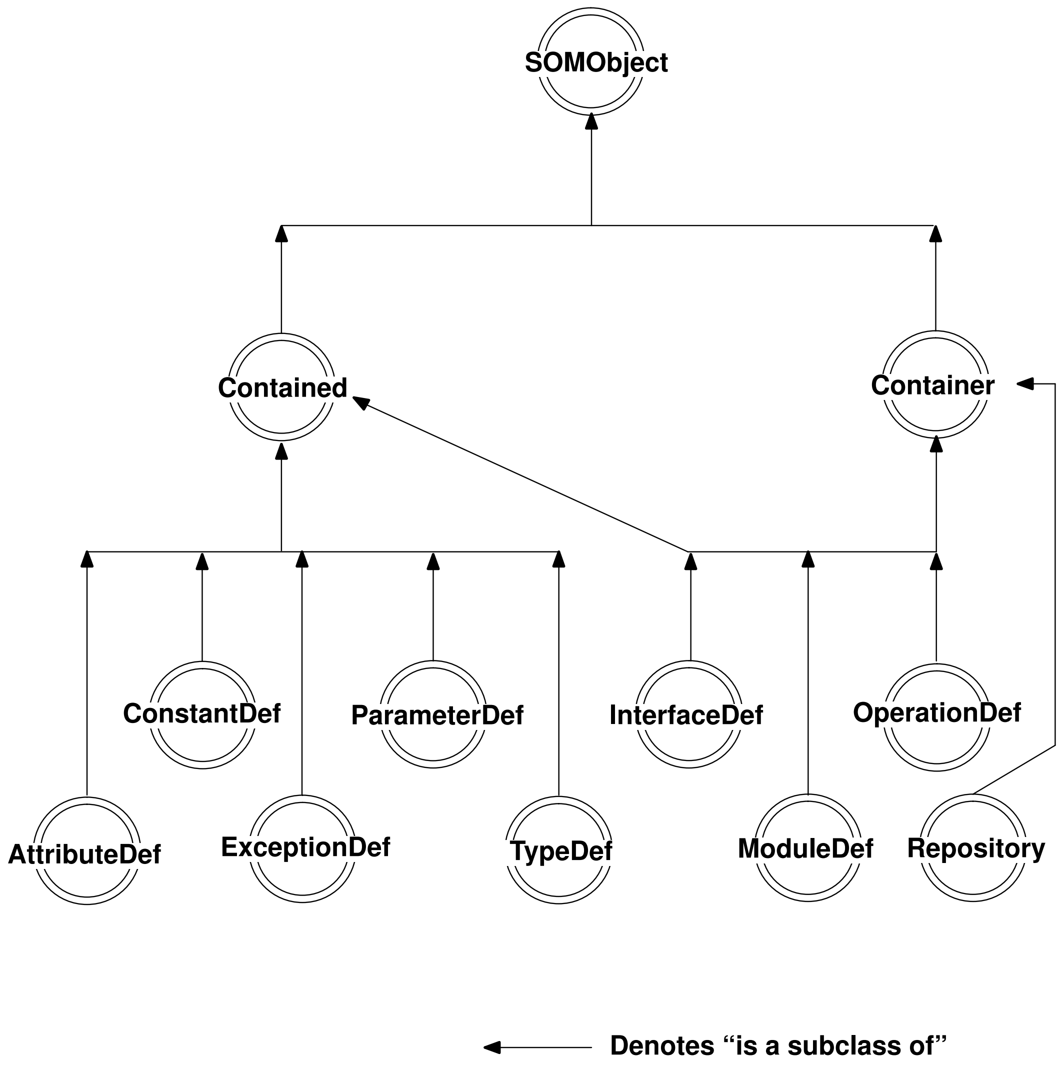
Original Class

SOMOA

Related Information

Methods: create_SOM_ref, **is_SOM_ref**

Chapter 3. Interface Repository Framework Reference



Interface Repository Framework Class Organization

AttributeDef Class

Description

The **AttributeDef** class provides the interface for **attribute** definitions in the Interface Repository.

File Stem

attribdf

Base

Contained

Metaclass

SOMClass

Ancestor Classes

Contained

SOMObject

Types

```
enum  AttributeMode  {NORMAL, READONLY};

struct  AttributeDescription   {
           Identifier          name;
           RepositoryId        id;
           RepositoryId        defined_in;
           TypeCode            type;
           AttributeMode       mode;
};
```

The **describe** method, inherited from **Contained**, returns an **AttributeDescription** structure in the *value* member of the **Description** structure (defined in the **Contained** class).

Attributes

Following is a list of each available attribute, with its corresponding type in parentheses, followed by a description of its purpose:

type (**TypeCode**)

The **TypeCode** that represents the type of the **attribute**. The **TypeCode** returned by the "_get_" form of the **type** attribute is contained in the receiving **AttributeDef** object, which retains ownership. Thus, the returned **TypeCode** should not be freed. To obtain a separate copy, use the **TypeCode_copy** operation. The "_set_" form of the attribute makes a private copy of the **TypeCode** you supply, to keep in the receiving object. You retain ownership of the passed **TypeCode**.

mode (**AttributeMode**)

The **AttributeMode** of the **attribute** (NORMAL or READONLY).

New Methods

None.

Overriding Methods

somInit

somUninit

somDumpSelf

somDumpSelfInt

describe

ConstantDef Class

Description

The **ConstantDef** class provides the interface for **constant** definitions in the Interface Repository.

File Stem

constdef

Base

Contained

Metaclass

SOMClass

Ancestor Classes

Contained

SOMObject

Types

```
struct ConstantDescription   {
          Identifier        name;
          RepositoryId      id;
          RepositoryId      defined_in;
          TypeCode          type;
          any               value;
};
```

The **describe** method, inherited from **Contained**, returns a **ConstantDescription** structure in the *value* member of the **Description** structure (defined in the **Contained** class).

Attributes

Following is a list of each available attribute, with its corresponding type in parentheses, followed by a description of its purpose:

type (**TypeCode**)
: The **TypeCode** that represents the type of **constant**.The **TypeCode** returned by the "_get_" form of the **type** attribute is contained in the receiving **ConstantDef** object, which retains ownership. Thus, the returned **TypeCode** should not be freed. To obtain a separate copy, use the **TypeCode_copy** operation. The "_set_" form of the attribute makes a private copy of the **TypeCode** you supply, to keep in the receiving object. You retain ownership of the passed **TypeCode**.

value (**any**)
: The value of the **constant**.

New Methods

None.

Overriding Methods

somInit

somUninit

somDumpSelf

somDumpSelfInt

describe

Contained Class

Description

The **Contained** class is the most generic form of interface for objects in SOM's CORBA-compliant Interface Repository (IR). All objects contained in the IR inherit this interface.

File Stem

containd

Base

SOMObject

Metaclass

SOMClass

Ancestor Classes

SOMObject

Types

```
typedef  string  RepositoryId;
struct  Description  {
            Identifier          name;
            any                 value;
};
```

Attributes

All attributes of the **Contained** class provide access to information kept within the receiving object. The "_get_" form of the attribute returns a memory reference that is only valid as long as the receiving object has not been freed (using **_somFree**). The "_set_" form of the attribute makes a (deep) copy of your data and places it in the receiving object. You retain ownership of all memory references passed using the "_set_" attributes.

Following is a list of each available attribute, with its corresponding type in parentheses, followed by a description of its purpose:

name (**Identifier**)

A simple name that identifies the **Contained** object within its containment hierarchy.

The name may not be unique outside of the containment hierarchy; thus it may require qualification by **ModuleDef** name and/or **InterfaceDef** name.

id (**RepositoryId**)

The value of the *id* field of the **Contained** object. This is a string that uniquely identifies any object in the IR; thus it needs no qualification. Note that **RepositoryId**s have no relationship to the SOM type **somId**.

defined_in (**RepositoryId**)

The value of the *defined_in* field of the **Contained** object. This ID uniquely identifies the container where the **Contained** object is defined. Objects without global scope that do not appear within any other object are, by default, placed in the Repository object.

somModifiers (**sequence<somModifier>**)

The somModifiers attribute is a sequence containing all modifiers associated with the object in the “implementation” section of the SOM IDL file where the receiving object is defined.

Note: This attribute is a SOM-unique extension of the Interface Repository; it is not stipulated by the CORBA specification.

New Methods

describe

within

Overriding Methods

somFree

somInit

somUninit

somDumpSelf

somDumpSelfInt

describe Method

Purpose

Returns a structure containing information defined in the IDL specification that corresponds to a specified **Contained** object in the Interface Repository.

IDL Syntax

Description describe ( );

Description

The **describe** method returns a structure containing information defined in the IDL specification of a **Contained** object. The specified object represents a component of an IDL interface (class) definition maintained within the Interface Repository.

When finished using the information in the returned **Description** structure, the client code must release the storage allocated for it. To free the associated storage, use a call similar to this:

```
if (desc.value._value)
     SOMFree (desc.value._value);
```

Warning: The **describe** method returns pointers to elements within objects (for example, **name**). Thus, the **somFree** method should *not* be used to release any of these objects while the **describe** information is still needed.

Parameters

receiver	A pointer to the **Contained** object in the Interface Repository for which a **Description** is needed**.**
ev	A pointer to the **Environment** structure for the caller.

Return Value

The **describe** method returns a structure of type **Description** containing information defined in the IDL specification of the receiving object.

The *name* field of the **Description** is the name of the type of description. The *name* values are from the following set:

{“ModuleDescription”, “InterfaceDescription”, “AttributeDescription”, “OperationDescription”, “ParameterDescription”, “TypeDescription”, “ConstantDescription”, “ExceptionDescription”}

The *value* field is a structure of type **any** whose *value* field is a pointer to a structure of the type named by the *name* field of the **Description.** This structure provides all of the information contained in the IDL specification of the *receiver*. For example, if the **describe** method is invoked on an object of type **AttributeDef**, the *name* field of the returned **Description** will contain the identifier “AttributeDescription” and the *value* field will contain an **any** structure whose *value* field is a pointer to an **AttributeDescription** structure.

Example

Here is a code fragment written in C that uses the **describe** method:

```
#include <containd.h>
#include <attribdf.h>
#include <somtc.h>

. . .

AttributeDef attr; /* An AttributeDef object (also a Contained) */
Description desc; /* .value field will be an AttributeDescription
*/
AttributeDescription *ad;
Environment *ev;

. . .

desc = Contained_describe (attr, ev);
ad = (AttributeDescription *) desc.value._value;
printf (”Attribute name: %s, defined in: %s\n”,
         ad->name, ad->defined_in);
printf (”Attribute type: ”);
TypeCode_print (ad->type, ev);
printf (”Attribute mode: %s\n”, ad->mode == AttributeDef_READONLY ?
        ”READONLY” : ”NORMAL”);
SOMFree (desc.value._value); /* Finished with describe output */
SOMObject_somFree (attr);    /* Finished with AttributeDef object
*/
```

Original Class

Contained

Related Information

Methods: within

within Method

Purpose

Returns a list of objects (in the Interface Repository) that contain a specified **Contained** object.

IDL Syntax

sequence<Container> within ( );

Description

The **within** method returns a sequence of objects within the Interface Repository that contain the specified **Contained** object. If the receiving object is an **InterfaceDef** or **ModuleDef**, it can only be contained by the object that defines it. Other objects can be contained by objects that define or inherit them.

If the object is global in scope, the sequence returned by **within** will have its **_length** field set to zero.

When finished using the sequence returned by this method, the client code is responsible for releasing each of the **Container**s in the sequence and freeing the sequence buffer. In C, this can be accomplished as follows:

```
if (seq._length) {
    long i;
    for (i=0; i<seq._length; i++)
         _somFree (seq._buffer[i]); /* Release each Container obj
*/
    SOMFree (seq._buffer);        /* Release the sequence buffer */
}
```

Parameters

receiver	A pointer to a **Contained** object for which containing objects are needed**.**
ev	A pointer to the **Environment** structure for the caller.

Return Value

The **within** method returns a sequence of **Container** objects that contain the specified **Contained** object.

Example

Here is a code fragment written in C that uses the **within** method:

```
#include <containd.h>
#include <containr.h>

. . .

Contained anObj;
Environment *ev;
sequence(Container) sc;
long i;
. . .

sc = Contained_within (anObj, ev);
printf (”%s is contained in (or inherited by):\n”,
     Contained__get_name (anObj, ev));
for (i=0; i<sc._length; i++) {
     printf (”\t%s\n”,
         Contained__get_name ((Contained) sc._buffer[i], ev));
     SOMObject_somFree (sc._buffer[i]);
}
if (sc._length)
     SOMFree (sc._buffer);
```

Original Class

Contained

Related Information

Methods: describe

Container Class

Description

The **Container** class is a generic interface that is common to all of the SOM CORBA-compliant Interface Repository (IR) objects that can hold or contain other objects. A **Container** object can be one of three types: **ModuleDef**, **InterfaceDef**, or **OperationDef.**

File Stem

containr

Base

SOMObject

Metaclass

SOMClass

Ancestor Classes

SOMObject

Types

```
typedef  string  InterfaceName;
// Valid values for InterfaceName are limited to the following set:
//   {"AttributeDef", "ConstantDef", "ExceptionDef", "InterfaceDef",
//     "ModuleDef", "ParameterDef", "OperationDef", "TypeDef", "all"}

struct  ContainerDescription  {
   Contained  *contained_object;
    Identifier  name;
    any   value;
};
```

New Methods

contents

describe_contents

lookup_name

Overriding Methods

somInit

somUninit

somDumpSelf

somDumpSelfInt

contents Method

Purpose

Returns a sequence indicating the objects contained within a specified **Container** object of the Interface Repository.

IDL Syntax

sequence<Contained> contents (
in InterfaceName *limit_type*,
in boolean *exclude_inherited***);**

Description

The **contents** method returns a list of objects contained by the specified **Container** object. Each object represents a component of an IDL interface (class) definition maintained within the Interface Repository.

The **contents** method is used to navigate through the hierarchy of objects within the Interface Repository: starting with the **Repository** object, this method can list all of the objects in the Repository, then all of the objects within the **ModuleDef** objects, then all within the **InterfaceDef** objects, and so on.

If the *limit_type* is set to "all", objects of all interface types are returned; otherwise, only objects of the requested interface type are returned. Valid values for **InterfaceName** are limited to the following set:

{"AttributeDef", "ConstantDef", "ExceptionDef", InterfaceDef", "ModuleDef", "ParameterDef", "OperationDef", "TypeDef", "all"}

If *exclude_inherited* is set to TRUE, any inherited objects will *not* be returned.

When finished using the sequence returned by this method, the client code is responsible for releasing each of the objects in the sequence and freeing the sequence buffer. In C, this can be accomplished as follows:

```
if (seq._length) {
    long i;
    for (i=0; i<seq._length; i++)
       SOMObject_somFree (seq._buffer[i]); /* Release each object
*/
    SOMFree (seq._buffer);                 /* Release the buffer
*/
}
```

Parameters

receiver	A pointer to a **Container** object whose contained objects are needed**.**
ev	A pointer to the **Environment** structure for the caller.
limit_type	The name of one interface type (see the previous list of valid types) or "all", to specify what type of objects the **contents** method should search for.
exclude_inherited	A **boolean** value: TRUE to exclude any inherited objects, or FALSE to include all objects.

Return Value

The **contents** method returns a sequence of pointers to objects contained within the specified **Container** object.

Example

Here is a code fragment written in C that uses the **contents** method:

```
#include <containr.h>

...

Container anObj;
Environment *ev;
sequence(Contained) sc;
long i;

...

sc = Container_contents (anObj, ev, ”all”, TRUE);
printf (”%s contains the following objects:\n”,
    SOMObject_somIsA (anObj, _Contained) ?
        Contained__get_name ((Contained) anObj, ev) :
        ”The Interface Repository”);
for (i=0; i<sc._length; i++) {
    printf (”\t%s\n”,
        Contained__get_name (sc._buffer[i], ev));
    SOMObject_somFree (sc._buffer[i]);
}
if (sc._length)
    SOMFree (sc._buffer);
else
    printf (”\t[none]\n”);
```

Original Class

Container

Related Information

Methods: describe_contents, **lookup_name**

describe_contents Method

Purpose

Returns a sequence of descriptions of the objects contained within a specified **Container** object of the Interface Repository.

IDL Syntax

sequence<ContainerDescription> describe_contents (
in InterfaceName *limit_type*,
in boolean *exclude_inherited,*
in long *max_returned_objs***);**

Description

The **describe_contents** method combines the operations of the **contents** method and the **describe** method. That is, for each object returned by the **contents** operation, the description of the object is returned by invoking its **describe** operation. Each object represents a component of an IDL interface (class) definition maintained within the Interface Repository.

If the *limit_type* is set to "all", objects of all interface types are returned; otherwise, only objects of the requested interface type are returned. Valid values for **InterfaceName** are limited to the following set:

{"AttributeDef", "ConstantDef", "ExceptionDef", "InterfaceDef", "ModuleDef", "ParameterDef", "OperationDef", "TypeDef", "all"}

If *exclude_inherited* is set to TRUE, any inherited objects will *not* be returned.

The *max_returned_objs* argument is used to limit the number of objects that can be returned. If *max_returned_objs* is set to –1, the results for all contained objects will be returned.

When finished using the sequence returned by this method, the client code is responsible for freeing the *value._value* field in each description, releasing each of the objects in the sequence, and freeing the sequence buffer. In C, this can be accomplished as follows:

```
if (seq._length) {
    long i;
    for (i=0; i<seq._length; i++) {
        if (seq._buffer[i].value._value)
                                    /* Release each description */
            SOMFree (seq._buffer[i].value._value);
        SOMObject_somFree (seq._buffer[i].contained_object);
                                      /* Release each object */
    }
    SOMFree (seq._buffer);            /* Release the buffer  */
}
```

Parameters

receiver	A pointer to a **Container** object whose contained object descriptions are needed.
ev	A pointer to the **Environment** structure for the caller.
limit_type	The name of one interface type (see the previous valid list) or "all", to specify what type of objects the **describe_contents** method should return.

exclude_inherited
A **boolean** value: TRUE to exclude any inherited objects, or FALSE to include all objects.

max_returned_objs
A **long** integer indicating the maximum number of objects to be returned by the method, or –1 to indicate no limit is set.

Return Value

The **describe_contents** method returns a sequence of **ContainerDescription** structures, one for each object contained within the specified **Container** object. Each **ContainerDescription** structure has a *contained_object* field, which points to the contained object, as well as *name* and *value* fields, which are the result of the **describe** method.

Example

Here is a code fragment written in C that uses the **describe_contents** method:

```
#include <containr.h>

...

Container anObj;
Environment *ev;
sequence(ContainerDescription) sc;
long i;

...

sc = Container_describe_contents (anObj, ev, ”all”, FALSE, -1L);
printf (”%s defines or inherits the following objects:\n”,
    SOMObject_somIsA (anObj, _Contained) ?
        Contained__get_name ((Contained) anObj, ev) :
        ”The Interface Repository”);
for (i=0; i<sc._length; i++) {
    printf (”\t%s\n”, sc._buffer[i].name);
    if (sc._buffer[i].value._value)
        SOMFree (sc._buffer[i].value._value);
    SOMObject_somFree (sc._buffer[i].contained_object);
}
if (sc._length)
    SOMFree (sc._buffer);
else
    printf (”\t[none]\n”);
```

Original Class

Container

Related Information

Methods: contents, **describe**, **lookup_name**

lookup_name Method

Purpose

Locates an object by name within a specified **Container** object of the Interface Repository, or within objects contained in the **Container** object.

IDL Syntax

sequence<Contained> lookup_name (
  in Identifier *search_name*,
  in long *levels_to_search*,
  in InterfaceName *limit_type*,
  in boolean *exclude_inherited***);**

Description

The **lookup_name** method locates an object by name within a specified **Container** object, or within objects contained in the **Container** object. The *search_name* parameter specifies the name of the object to be found. Each object represents a component of an IDL interface (class) definition maintained within the Interface Repository.

The *levels_to_search* argument controls whether the lookup is constrained to the specified **Container** object or whether objects contained within the **Container** object are also searched. The *levels_to_search* value should be –1 to search the **Container** and all contained objects; it should be 1 to search only the **Container** itself.

If *limit_type* is set to "all", the lookup locates an object of the specified name with any interface type; otherwise, the search locates the object only if it has the designated interface type. Valid values for **InterfaceName** are limited to the following set:

{"AttributeDef", "ConstantDef", "ExceptionDef", "InterfaceDef", "ModuleDef", "ParameterDef", "OperationDef", "TypeDef", "all"}

If *exclude_inherited* is set to TRUE, any inherited objects will *not* be returned.

When finished using the sequence returned by this method, the client code is responsible for releasing each of the objects in the sequence and freeing the sequence buffer. In C, this can be accomplished as follows:

```
if (seq._length) {
    long i;
    for (i=0; i<seq._length; i++)
        SOMObject_somFree (seq._buffer[i]);
                                        /* Release each object */
    SOMFree (seq._buffer);              /* Release the buffer  */
    }
```

Parameters

receiver A pointer to a **Container** object in which to locate the object**.**

ev A pointer to the **Environment** structure for the caller.

search_name The name of the object to be located.

levels_to_search
A long having the value 1 or –1.

limit_type The name of one interface type (see the previous list of valid items) specify what type of object to search for.

exclude_inherited
A **boolean** value: TRUE to exclude an object when it is inherited, or FALSE to return the object from wherever it is found.

Return Value

The **lookup_name** method returns a sequence of pointers to objects of the given name contained within the specified **Container** object, or within objects contained in the **Container** object.

Example

Here is a code fragment written in C that uses the **lookup_name** method:

```
#include <containr.h>
#include <containd.h>
#include <repostry.h>

...

Container repo;
Environment *ev;
sequence(Contained) sc;
long i;
Identifier nameToFind;

...

repo = (Container) RepositoryNew ();
sc = Container_lookup_name (repo, ev, nameToFind, -1, "all",
TRUE);
printf ("%d object%s found:\n",
         sc._length, sc._length == 1 ? "" : "s");
for (i=0; i<sc._length; i++) {
    printf ("\t%s\n",
        Contained__get_id (sc._buffer[i], ev));
    SOMObject_somFree (sc._buffer[i]);
}
if (sc._length)
    SOMFree (sc._buffer);
```

Original Class

Container

Related Information

Methods: contents, **describe_contents**

ExceptionDef Class

Description

The **ExceptionDef** class provides the interface for **exception** definitions in the Interface Repository.

File Stem

excptdef

Base

Contained

Metaclass

SOMClass

Ancestor Classes

Contained

SOMObject

Types

```
struct ExceptionDescription   {
            Identifier          name;
            RepositoryId        id;
            RepositoryId        defined_in;
            TypeCode            type;
};
```

The **describe** method, inherited from **Contained**, returns an **ExceptionDescription** structure in the *value* member of the **Description** structure (defined in the **Contained** class).

Attributes

Following is a list of each available attribute, with its corresponding type in parentheses, followed by a description of its purpose:

type (**TypeCode**)
: The **TypeCode** that represents the type of the **exception**. The **TypeCode** returned by the "_get_" form of the **type** attribute is contained in the receiving **ExceptionDef** object, which retains ownership. Thus the returned **TypeCode** should not be freed. To obtain a separate copy, use the **TypeCode_copy** operation. The "_set_" form of the attribute makes a private copy of the **TypeCode** you supply, to keep in the receiving object. You retain ownership of the passed **TypeCode**.

New Methods

None.

Overriding Methods

somInit

somUninit

somDumpSelf

somDumpSelfInt

describe

InterfaceDef Class

Description

The **InterfaceDef** class provides the interface for **interface** definitions in the Interface Repository.

File Stem

intfacdf

Base

Contained, **Container**

Metaclass

SOMClass

Ancestor Classes

Contained

Container

SOMObject

Types

```
struct FullInterfaceDescription   {
        Identifier          name;
        RepositoryId        id;
        RepositoryId        defined_in;
        sequence<OperationDef::OperationDescription>  operation;
        sequence<AttributeDef::AttributeDescription>  attributes;
};
```

```
struct InterfaceDescription   {
        Identifier          name;
        RepositoryId        id;
        RepositoryId        defined_in;
};
```

The **describe** method, inherited from **Contained**, returns an **InterfaceDescription** structure in the *value* member of the **Description** structure (defined in the **Contained** class). The **describe_contents** method, inherited from **Container**, returns a sequence of these **Description** structures, each carrying a reference to an **InterfaceDescription** structure in its *value* member.

Implementation note: The two sequences “OperationDescription” and “AttributeDescription” are built dynamically within the **FullInterfaceDescription** structure, due to the **InterfaceDef** class’s inheritance from the **Contained** class.

Attributes

All attributes of the **InterfaceDef** class provide access to information kept within the receiving **InterfaceDef** object. The “_get_” form of the attribute returns a memory reference that is only valid as long as the receiving object has not been freed (using **_somFree**). The “_set_” form of the attribute makes a (deep) copy of your data and places it in the receiving **InterfaceDef** object. You retain ownership of all memory references passed using the “_set_” attribute forms.

Following is a list of each available attribute, with its corresponding type in parentheses, followed by a description of its purpose:

base_interfaces (**sequence<RepositoryId>**)
: The sequence of RepositoryIds for all of the interfaces that the receiving interface inherits.

instanceData (**TypeCode**)
: The TypeCode of a structure whose members are the internal instance variables, if any, described in the SOM implementation section of the interface.

  Note: This attribute is a SOM-unique extension of the Interface Repository; it is not stipulated by the CORBA specifications.

New Methods

describe_interface

Overriding Methods

somInit

somUninit

somDumpSelf

somDumpSelfInt

describe

within

describe_interface Method

Purpose

Returns (from the Interface Repository) a description of all the methods and attributes of an interface definition.

IDL Syntax

FullInterfaceDescription describe_interface ( );

Description

The **describe_interface** method returns a description of all the methods and attributes of an interface definition that are held in the Interface Repository.

When finished using the **FullInterfaceDescription** returned by this method, the client code is responsible for freeing the _buffer fields of the two sequences it contains. In C, this can be accomplished as follows:

```
if (fid.operation._length)
    SOMFree (fid.operation._buffer);        /* Release the buffer
*/
if (fid.attributes._length)
    SOMFree (fid.attributes._buffer);       /* Release the buffer
*/
```

Parameters

receiver	A pointer to an object of class **InterfaceDef** representing the Interface Repository object where an interface definition is stored.
ev	A pointer where the method can return exception information if an error is encountered.

Return Value

The **describe_interface** method returns a description of all the methods and attributes of an interface definition that are held in the Interface Repository.

Example

Here is a code fragment written in C that uses the **describe_interface** method:

```
#include <intfacdf.h>

...

InterfaceDef idef;
Environment *ev;
FullInterfaceDescription fid;
long i;

...

fid = InterfaceDef_describe_interface (idef, ev);
printf (”The %s interface has the following attributes:\n”,
    Contained__get_name ((Contained) idef, ev));
if (!fid.attributes._length)
    printf (”\t[none]\n”);
else {
    for (i=0; i<fid.attributes._length; i++)
        printf (”\t%s\n”, fid.attributes._buffer[i].name);
    SOMFree (fid.attributes._buffer);
}

printf (”and the following methods:\n”)
if (!fid.operation._length)
    printf (”\t[none]\n”);
else {
    for (i=0; i<fid.operation._length; i++)
        printf (”\t%s\n”, fid.operation._buffer[i].name);
    SOMFree (fid.operation._buffer);
}
```

Original Class

InterfaceDef

ModuleDef Class

Description

The **ModuleDef** class provides the interface for **module** definitions in the Interface Repository.

File Stem

moduledf

Base

Contained, **Container**

Metaclass

SOMClass

Ancestor Classes

Contained

Container

SOMObject

Types

```
struct ModuleDescription   {
            Identifier           name;
            RepositoryId         id;
            RepositoryId         defined_in;
};
```

The **describe** method, inherited from **Contained**, returns a **ModuleDescription** structure in the *value* member of the **Description** structure (defined in the **Contained** class). The **describe_contents** method, inherited from **Container**, returns a sequence of these **Description** structures, each carrying a reference to a **ModuleDescription** structure in its *value* member.

New Methods

None.

Overriding Methods

somInit

somUninit

somDumpSelf

somDumpSelfInt

describe

within

OperationDef Class

Description

The **OperationDef** class provides the interface for operation (method) definitions in the Interface Repository.

File Stem

operatdf

Base

Contained, **Container**

Metaclass

SOMClass

Ancestor Classes

Contained

Container

SOMObject

Types

```
typedef  Identifier  ContextIdentifier;
enum  OperationMode  {NORMAL, ONEWAY};

struct  OperationDescription  {
          Identifier           name;
          RepositoryId         id;
          RepositoryId         defined_in;
          TypeCode             result;
          OperationMode        mode;
          sequence<ContextIdentifier>  contexts;
          sequence<ParameterDef::ParameterDescription>  parameter;
          sequence<ExceptionDef::ExceptionDescription>  exceptions;
};
```

The **describe** method, inherited from **Contained**, returns an **OperationDescription** structure in the *value* member of the **Description** structure (defined in the **Contained** class). The **describe_contents** method, inherited from **Container**, returns a sequence of these **Description** structures, each carrying a reference to an **OperationDescription** structure in its *value* member.

Attributes

Following is a list of each available attribute, with its corresponding type in parentheses, followed by a description of its purpose:

result (**TypeCode**)

The **TypeCode** that represents the type of the operation (method). The **TypeCode** returned by the "_get_" form of the **type** attribute is contained in the receiving **OperationDef** object, which retains ownership. Thus the returned **TypeCode** should not be freed. To obtain a separate copy, use the **TypeCode_copy** operation. The "_set_" form of the attribute makes a private copy of the **TypeCode** you supply, to keep in the receiving object. You retain ownership of the passed **TypeCode**.

mode (**OperationMode**)

The *OperationMode* of the operation (method), either NORMAL or ONEWAY.

contexts (**sequence<ContextIdentifier>**)

The list of *ContextIdentifiers* associated with the operation (method). The "_get_" form of the attribute returns a sequence whose buffer is owned by the receiving *OperationDef* object. You should not free it. The "_set_" form of the attribute makes a (deep) copy of the passed sequence; you retain ownership of the original storage.

New Methods

None.

Overriding Methods

somInit

somUninit

somDumpSelf

somDumpSelfInt

describe

ParameterDef Class

Description

The **ParameterDef** class provides the interface for **parameter** definitions in the Interface Repository.

File Stem

paramdef

Base

Contained

Metaclass

SOMClass

Ancestor Classes

Contained

SOMObject

Types

```
enum  ParameterMode  {IN, OUT, INOUT};

struct  ParameterDescription   {
            Identifier           name;
            RepositoryId         id;
            RepositoryId         defined_in;
            TypeCode             type;
            ParameterMode        mode;
};
```

The **describe** method, inherited from **Contained**, returns a **ParameterDescription** structure in the *value* member of the **Description** structure (defined in the **Contained** class).

Attributes

Following is a list of each available attribute, with its corresponding type in parentheses, followed by a description of its purpose:

type (**TypeCode**)
: The **TypeCode** that represents the type of the parameter. The **TypeCode** returned by the "_get_" form of the **type** attribute is contained in the receiving **ParameterDef** object, which retains ownership. Hence, the returned **TypeCode** should not be freed. To obtain a separate copy, use the **TypeCode_copy** operation. The "_set_" form of the attribute makes a private copy of the **TypeCode** you supply, to keep in the receiving object. You retain ownership of the passed **TypeCode**.

mode (**ParameterMode**)
: The ParameterMode of the parameter (IN, OUT, or INOUT).

New Methods

None.

Overriding Methods

somInit

somUninit

somDumpSelf

somDumpSelfInt

describe

Repository Class

Description

The **Repository** class provides global access to SOM's CORBA-compliant Interface Repository (IR), which is discussed in Chapter 7, "The Interface Repository Framework," of the *SOM Toolkit User's Guide.*

File Stem

repostry

Base

Container

Metaclass

SOMClass

Ancestor Classes

SOMObject

Types

```
struct  RepositoryDescription  {
            Identifier          name;
            RepositoryId        id;
            RepositoryId        defined_in;
};
```

The inherited **describe_contents** method returns an instance of the **RepositoryDescription** structure in the *value* member of the **Description** structure defined in the **Container** interface.

New Methods

lookup_id

lookup_modifier

release_cache

Overriding Methods

describe_contents

somInit

somUninit

somFree

somDumpSelf

somDumpSelfInt

lookup_id Method

Purpose

Returns the object having a specified **RepositoryId**.

IDL Syntax

Contained lookup_id (
in RepositoryId *search_id***);**

Description

The **lookup_id** method returns the object having a **RepositoryId** given by the specified *search_id* argument. The returned object represents a component of an IDL interface (class) definition maintained within the Interface Repository.

When finished using the object returned by this method, the client code is responsible for releasing it, using the **somFree** method.

Parameters

receiver	A pointer to an object of class **Repository** representing SOM's Interface Repository**.**
ev	A pointer where the method can return exception information if an error is encountered.
search_id	An ID value of type **RepositoryId** that uniquely identifies the desired object in the Interface Repository.

Return Value

The **lookup_id** method returns the **Contained** object that has the specified **RepositoryId**.

Example

Here is a code fragment written in C that uses the **lookup_id** method:

```
#include <containd.h>
#include <repostry.h>

...

Repository repo;
Environment *ev;
Contained c;
RepositoryId objectToFind;

...

repo = RepositoryNew ();
c = Repository_lookup_id (repo, ev, objectToFind);
if (c) {
    printf (”lookup_id found object of type: %s, named: %s\n”,
        SOMObject_somGetClassName (c), Contained__get_name (c,
ev));
    SOMObject_somFree (c);
}
```

Original Class

Repository

Related Information

Methods: lookup_modifier, **lookup_name**, **contents**, **within**

lookup_modifier Method

Purpose

Returns the value of a given SOM **modifier** for a specified object [that is, for an object that is a component of an IDL interface (class) definition maintained within the Interface Repository].

IDL Syntax

string lookup_modifier (
                in RepositoryId *id***,**
                in string *modifier***);**

Description

The **lookup_modifier** method returns the string value of the given SOM **modifier** for an object with the specified **RepositoryId** within the Interface Repository. For a discussion of SOM modifiers, see the topic “Modifier statements” in Chapter 4, “Implementing SOM Classes,” of the *SOM Toolkit User’s Guide.*

If the object with the given **RepositoryId** does not exist or does not possess the modifier, then NULL (or zero) is returned. If the object exists but the specified modifier does not have a value, a zero-length string value is returned.

Note: The **lookup_modifier** method is *not* stipulated by the CORBA specifications; it is a SOM-unique extension to the Interface Repository.

Parameters

receiver	A pointer to an object of class **Repository** representing SOM’s Interface Repository.
ev	A pointer where the method can return exception information if an error is encountered.
id	The **RepositoryId** of the object whose modifier value is needed.
modifier	The name of a specific (SOM or user-specified) modifier whose string value is needed.

Return Value

The **lookup_modifier** method returns the string value of the given SOM **modifier** for an object with the specified **RepositoryId**, if it exists. If an existing modifier has no value, a zero-length string value is returned. If the object cannot be found, then NULL (or zero) is returned.

When the string value is no longer needed, client code must free the space for the string (using **SOMFree**).

Example

Here is a code fragment written in C that uses the **lookup_modifier** method:

```
#include <repostry.h>

...

Repository repo;
Environment *ev;
RepositoryId objectId;
string filestem;i

...

repo = RepositoryNew ();
filestem = Repository_lookup_modifier (repo, ev, objectId,
                                                  ”filestem”);
if (filestem) {
    printf
         (”The %s object’s filestem modifier has the value
\”%s\”\n”,
            objectId, filestem);
    SOMFree (filestem);
} else
    printf (”No filestem modifier could be found for %s\n”,
            objectId);
```

Original Class

Repository

Related Information

Methods: lookup_id, **lookup_name**

release_cache Method

Purpose

Permits the Repository object to release the memory occupied by Interface Repository objects that have been implicitly referenced.

Syntax

void release_cache ( );

Description

This method allows the Repository object to release the memory occupied by implicitly referenced Interface Repository objects. Some methods (such as **describe_contents** and **lookup_name**) may cause some objects to be instantiated that are not directly accessible through object references that have been returned to the user. These objects are kept in an internal Interface Repository cache until the **release_cache** method is used to free them. The internal cache continuously replenishes itself over time as the need arises.

Parameters

receiver	A pointer to an object of class **Repository** representing SOM's Interface Repository.
ev	A pointer where the method can return exception information if an error is encountered.

Example

```
#include <repostry.h>

...

Repository repo;
Environment *ev;
sequence(ContainerDescription) scd;

...

scd = Container_describe_contents (
      (Container) repo, ev, ”TypeDef”, TRUE, –1);
Repository_release_cache (repo, ev);
```

Original Class

Repository

Related Information

See the section entitled “A word about memory management” in Chapter 7 of the *SOM Toolkit User's Guide.*

TypeDef Class

Description

The **TypeDef** class provides the interface for **typedef** definitions in the Interface Repository.

File Stem

typedef

Base

Contained

Metaclass

SOMClass

Ancestor Classes

Contained

SOMObject

Types

```
struct TypeDescription   {
          Identifier          name;
          RepositoryId        id;
          RepositoryId        defined_in;
          TypeCode            type;
};
```

The **describe** method, inherited from **Contained**, returns a **TypeDescription** structure in the *value* member of the **Description** structure (defined in the **Contained** class).

Attributes

Following is a list of each available attribute, with its corresponding type in parentheses, followed by a description of its purpose:

type (**TypeCode**)

The **TypeCode** that represents the type of the **typedef**. The **TypeCode** returned by the "_get_" form of the **type** attribute is contained in the receiving **TypeDef** object, which retains ownership. Hence, the returned **TypeCode** should not be freed. To obtain a separate copy, use the **TypeCode_copy** operation. The "_set_" form of the attribute makes a private copy of the **TypeCode** you supply, to keep in the receiving object. You retain ownership of the passed **TypeCode**.

New Methods

None.

Overriding Methods

somInit

somUninit

somDumpSelf

somDumpSelfInt

describe

TypeCode_alignment Function

Purpose

Supplies the alignment value for a given **TypeCode**.

IDL Syntax

short TypeCode_alignment ( );

Description

This function returns the alignment information associated with the given **TypeCode**. The alignment value is a short integer that should evenly divide any memory address where an instance of the type described by the **TypeCode** will occur.

Parameters

tc	The **TypeCode** whose alignment information is desired.
ev	A pointer to an Environment structure.

Return Value

A short integer containing the alignment value.

Related Information

Functions: TypeCodeNew, **TypeCode_equal**, **TypeCode_free**, **TypeCode_kind**, **TypeCode_param_count**, **TypeCode_parameter**, **TypeCode_print**, **TypeCode_setAlignment**, **TypeCode_size**

TypeCode_copy Function

Purpose

Creates a new copy of a given **TypeCode**.

IDL Syntax

TypeCode TypeCode_copy ( );

Description

The **TypeCode_copy** function creates a new copy of a given **TypeCode**. **TypeCode**s are complex data structures whose actual representation is hidden, and may contain internal references to **string**s and other **TypeCode**s. The copy created by this function is guaranteed not to refer to any previously existing **TypeCode**s or **string**s, and hence can be used long after the original **TypeCode** is freed or released (**TypeCode**s are typically contained in Interface Repository objects whose memory resources are released by the **_somFree** method).

All of the memory used to construct the **TypeCode** copy is allocated dynamically and should be subsequently freed *only* by using the **TypeCode_free** function.

This function is a SOM-unique extension to the CORBA standard.

Parameters

tc	The **TypeCode** to be copied.
ev	A pointer to an **Environment** structure. The CORBA standard mandates the use of this structure as a standard way to return exception information when an error condition is detected.

Return Value

A new **TypeCode** with no internal references to any previously existing **TypeCode**s or **string**s. If a copy cannot be created successfully, the value NULL is returned. No exceptions are raised by this function.

Related Information

Functions: TypeCodeNew, **TypeCode_alignment**, **TypeCode_equal**, **TypeCode_free**, **TypeCode_kind**, **TypeCode_param_count**, **TypeCode_parameter**, **TypeCode_print**, **TypeCode_setAlignment**, **TypeCode_size**

TypeCode_equal Function

Purpose

Compares two **TypeCode**s for equality.

IDL Syntax

boolean TypeCode_equal (
TypeCode *tc2***);**

Description

The **TypeCode_equal** function can be used to determine if two distinct **TypeCode**s describe the same underlying abstract data type.

Parameters

tc	One of the **TypeCode**s to be compared.
ev	A pointer to an **Environment** structure. The CORBA standard mandates the use of this structure as a standard way to return exception information when an error condition is detected.
tc2	The other **TypeCode** to be compared.

Return Value

Returns TRUE (1) if the **TypeCode**s *tc* and *tc2* describe the same data type, with the same alignment. Otherwise, FALSE (0) is returned. No exceptions are raised by this function.

Related Information

Functions: TypeCodeNew, **TypeCode_alignment**, **TypeCode_copy**, **TypeCode_free**, **TypeCode_kind**, **TypeCode_param_count**, **TypeCode_parameter**, **TypeCode_print**, **TypeCode_setAlignment**, **TypeCode_size**

TypeCode_free Function

Purpose

Destroys a given **TypeCode** by freeing all of the memory used to represent it.

IDL Syntax

void TypeCode_free ( );

Description

The **TypeCode_free** function destroys a given **TypeCode** by freeing all of the memory used to represent it. **TypeCode**s obtained from the **TypeCode_copy** or **TypeCodeNew** functions should be freed using **TypeCode_free**. **TypeCode**s contained in Interface Repository objects should never be freed. Their memory is released when a **_somFree** method releases the Interface Repository object.

The **TypeCode_free** operation has no effect on **TypeCode** constants. **TypeCode** constants are static **TypeCode**s declared in the header file **somtcnst.h** or generated in files emitted by the SOM Compiler. Since **TypeCode** constants may be used interchangeably with dynamically created **TypeCode**s, it is *not* considered an error to attempt to free a **TypeCode** constant with the **TypeCode_free** function.

This function is a SOM-unique extension to the CORBA standard.

Parameters

tc	The **TypeCode** to be freed.
ev	A pointer to an **Environment** structure. The CORBA standard mandates the use of this structure as a standard way to return exception information when an error condition is detected.

Return Value

None. No exceptions are raised by this function.

Related Information

Functions: TypeCodeNew, **TypeCode_alignment**, **TypeCode_copy**, **TypeCode_equal**, **TypeCode_kind**, **TypeCode_param_count**, **TypeCode_parameter**, **TypeCode_print**, **TypeCode_setAlignment**, **TypeCode_size**

TypeCode_kind Function

Purpose

Categorizes the abstract data type described by a **TypeCode**.

IDL Syntax

```
TCKind  TypeCode_kind ( );

enum  TCKind {
                  tk_null,  tk_void,
                  tk_short,  tk_long,  tk_ushort,  tk_ulong,
                  tk_float,  tk_double,  tk_boolean,  tk_char,
                  tk_octet,  tk_any,  tk_TypeCode,  tk_Principal,
                  tk_objref,  tk_struct,  tk_union,  tk_enum,  tk_string,
                  tk_sequence,  tk_array,  tk_pointer,  tk_self, tk_foreign
              };
```

Description

The **TypeCode_kind** function can be used to classify a **TypeCode** into one of the categories listed in the **TCKind** enumeration. Based on the “kind” classification, a **TypeCode** may contain 0 or more additional parameters to fully describe the underlying data type.

The following table indicates the number and function of these additional parameters. **TCKind** entries not listed in the table are basic data types and do not have any additional parameters. The designation “N” refers to the number of members in a **struct** or **union**, or the number of enumerators in an **enum**.

TypeCode Information per TCKind Category			
TCKind	**Parameters**	**Type**	**Function**
tk_objref	1	**string**	The ID of the corresponding **InterfaceDef** in the Interface Repository.
tk_struct	2N+1	**string**	The name of the **struct**.
		—next 2 repeat for each member—	
		string	The name of the **struct** member.
		TypeCode	The type of the **struct** member.
tk_union	3N+2	**string**	The name of the **union**.
		TypeCode	The type of the discriminator.
		—next 3 repeats for each enumerator—	
		long	The label value.
		string	The name of the member.
		TypeCode	The type of the member.
tk_enum	N+1	**string**	The name of the **enum**.
		——next repeats for each enumerator——	
		string	The name of the enumerator.
tk_string	1	**long**	The maximum string length or 0.
tk_sequence	2	**TypeCode**	The type of element in the sequence.
		long	The maximum number of elements or 0.
tk_array	2	**TypeCode**	The type of element in the **array**.
		long	The maximum number of elements.
tk_pointer*	1	**TypeCode**	The type of the referenced datum.
tk_self*	1	**string**	The name of the referenced enclosing **struct** or **union**.
tk_foreign*	3	**string**	The name of the foriegn type.
		string	The implementation context.
		long	The size of an instance.

Note: *The **TCKind** values **tk_pointer**, **tk_self**, and **tk_foreign** are SOM-unique extensions to the CORBA standard. They are provided to permit **TypeCode**s to describe types that cannot be expressed in standard IDL.

The **tk_pointer TypeCode** contains only one parameter—a **TypeCode** which describes the data type that the pointer references. The **tk_self TypeCode** is used to describe a “self-referential” structure or union without introducing unbounded recursion in the **TypeCode**. For example, the following C struct:

```
struct node {
   long count;
   struct node *next;
   };
```

could be described with a **TypeCode** created as follows:

```
TypeCode tcForNode;

tcForNode = TypeCodeNew (tk_struct, ”node”,
    ”count”, TypeCodeNew (tk_long),
    ”next”, TypeCodeNew (tk_pointer,
    TypeCodeNew (tk_self, ”node”)));
```

The **tk_foreign TypeCode** provides a more general escape mechanism, allowing **TypeCode**s to be created that partially describe non-IDL types. Since these foreign **TypeCode**s carry only a partial description of a type, the “implementation context” parameter can be used by a non-IDL execution environment to recognize other types that are known or understood in that environment. For more information about using foreign **TypeCode**s in SOM IDL files see the *SOM Toolkit User’s Guide*.

Note that the use of self-referential structures, pointers, or foreign types is beyond the scope of the CORBA standard, and may result in a loss of portability or distributability in client code.

Parameters

tc	The **TypeCode** whose **TCKind** categorization is requested.
ev	A pointer to an **Environment** structure. The CORBA standard mandates the use of this structure as a standard way to return exception information when an error condition is detected.

Return Value

Returns one of the enumerators listed in the **TCKind** enumeration shown previously. No exceptions are raised by this function.

Related Information

Functions: TypeCodeNew, **TypeCode_alignment**, **TypeCode_copy**, **TypeCode_equal**, **TypeCode_free**, **TypeCode_param_count**, **TypeCode_parameter**, **TypeCode_print**, **TypeCode_setAlignment**, **TypeCode_size**

TypeCodeNew Function

Purpose

Creates a new **TypeCode** instance.

Syntax

TypeCode TypeCodeNew (TCKind *tag*, ...**);**

[The actual parameters indicated by "..." are variable in number and type, depending on the value of the *tag* parameter.] There are *no* implicit parameters to this function.

TypeCodeNew (tk_objref, **string** *interfaceId***);**
TypeCodeNew (tk_string, **long** *maxLength***);**
TypeCodeNew (tk_sequence, **TypeCode** *seqTC*, **long** *maxLength***);**
TypeCodeNew (tk_array, **TypeCode** *arrayTC*, **long** *length***);**
TypeCodeNew (tk_pointer, **TypeCode** *ptrTC***);**
TypeCodeNew (tk_self, **string** *structOrUnionName***);**
TypeCodeNew (tk_foreign, **string** *typename*, **string** *impCtx*, **long** *instSize***);**

TypeCodeNew (tk_struct, **string** *name*,
string *mbrName*, **TypeCode** *mbrTC*, [...,]
[*mbrName* and *mbrTC* repeat as needed]
NULL);

TypeCodeNew (tk_union, **string** *name*, **TypeCode** *swTC*,
long *flag*, **long** *labelValue*, **string** *mbrName*, **TypeCode** *mbrTC*, [...,]
[*flag, labelValue, mbrName* and *mbrTC* repeat as needed]
NULL);

TypeCodeNew (tk_enum, **string** *name*,
string *enumId*, [...,]
[*enumId*s repeat as needed]
NULL);

TypeCodeNew (TCKind *allOtherTagValues***);**

Description

The **TypeCodeNew** function creates a new instance of a **TypeCode** from the supplied parameters. **TypeCode**s are complex data structures whose actual representation is hidden. The number and types of arguments required by **TypeCodeNew** varies depending on the value of the first argument. All of the valid invocation sequences are shown in the previous section. There are *no* implicit parameters to this function.

All **TypeCode**s created by **TypeCodeNew** should be destroyed (when no longer needed) using the **TypeCode_free** function.

This function is a SOM-unique extension to the CORBA standard.

Parameters

tag	The type or category of **TypeCode** to create.
interfaceId	A string containing the fully-qualified interface name that is the subject of an object reference type.
name	A string that gives the name of a **struct**, **union**, or **enum**.
mbrName	A string that gives the name of a **struct** or **union** member element.
enumId	A string that gives the name of an **enum** enumerator.

structOrUnionName
A string that gives the name of a **struct** or **union** that has been previously named in the current **TypeCode** and is the subject of a self-referential pointer type. See the footnote on **tk_self** in the table given in the **TypeCode_kind** function description for an example of what this means and how it is applied.

maxLength
The maximum permitted length of a **string** or a **sequence**. The value 0 (zero) means that the **string** or **sequence** is considered unbounded.

length
The maximum number of elements that can be stored in an array. All IDL arrays are bounded, hence a value of zero denotes an array of zero elements.

flag
One of the following constant values used to distinguish a labeled case in an IDL discriminated **union** switch statement from the default case:

TCREGULAR_CASE	The value 1
TCDEFAULT_CASE	The value 2

labelValue
The actual value associated with a regular labeled case in an IDL discriminated **union** switch statement. If preceded by the argument TCDEFAULT_CASE, the value zero should be used.

mbrTC
A **TypeCode** that represents the data type of a **struct** or **union** member.

swTC
A **TypeCode** that represents the data type of the discriminator in an IDL **union** statement.

seqTC
A **TypeCode** that describes the data type of the elements in a **sequence**.

arrayTC
A **TypeCode** that describes the data type of the elements of an **array**.

ptrTC
A **TypeCode** that describes the data type referenced by a pointer.

typename
A string that provides the name of a foreign type.

impCtx
A string that identifies an implementation context where a foreign type is understood.

instSize
A long that holds the size of a foreign type instance. If the size is variable or is not known, the value zero should be used.

allOtherTagValues
One of the values: **tk_null**, **tk_void**, **tk_short**, **tk_long**, **tk_ushort**, **tk_ulong**, **tk_float**, **tk_double**, **tk_boolean**, **tk_char**, **tk_octet**, **tk_any**, **tk_TypeCode**, or **tk_Principal**

All of these tags represent basic IDL data types that do not require any other descriptive parameters.

Return Value

A new **TypeCode** instance, or NULL if the new instance could not be created.

Related Information

Functions: TypeCode_alignment, **TypeCode_copy**, **TypeCode_equal**, **TypeCode_free**, **TypeCode_kind**, **TypeCode_param_count**, **TypeCode_paramater**, **TypeCode_print**, **TypeCode_setAlignment**, **TypeCode_size**

TypeCode_param_count Function

Purpose

Obtains the number of parameters available in a given **TypeCode**.

IDL Syntax

long TypeCode_param_count ( );

Description

The **TypeCode_param_count** function can be used to obtain the actual number of parameters contained in a specified **TypeCode**. Each **TypeCode** contains sufficient parameters to fully describe its underlying abstract data type. Refer to the table given in the description of the **TypeCode_kind** function.

Parameters

tc	The **TypeCode** whose parameter count is desired.
ev	A pointer to an **Environment** structure. The CORBA standard mandates the use of this structure as a standard way to return exception information when an error condition is detected.

Return Value

Returns the actual number of parameters associated with the given **TypeCode**, in accordance with the table shown in the **TypeCode_kind** description. No exceptions are raised by this function.

Related Information

Functions: TypeCodeNew, **TypeCode_alignment**, **TypeCode_copy**, **TypeCode_equal**, **TypeCode_free**, **TypeCode_kind**, **TypeCode_paramater**, **TypeCode_print**, **TypeCode_setAlignment**, **TypeCode_size**

TypeCode_parameter Function

Purpose

Obtains a specified parameter from a given **TypeCode**.

IDL Syntax

```
any  TypeCode_parameter (
                               long index);
```

Description

The **TypeCode_parameter** function can be used to obtain any of the parameters contained in a given **TypeCode**. Refer to the table shown in the description of the **TypeCode_kind** function for a list of the number and type of parameters associated with each category of **TypeCode**.

Parameters

tc	The **TypeCode** whose parameter is desired.
ev	A pointer to an **Environment** structure. The CORBA standard mandates the use of this structure as a standard way to return exception information when an error condition is detected.
index	The number of the desired parameter. Parameters are numbered from 0 to N–1, where N is the value returned by the **Typecode_param_count** function.

Return Value

Returns the requested parameter in the form of an **any**. This function raises the Bounds exception if the value of the index exceeds the number of parameters available in the given **TypeCode**. Because the values exist within the specified **TypeCode**, you should not free the results returned from this function.

An **any** is a basic IDL data type that is represented as the following structure in C or C++:

```
typedef struct any {
    TypeCode _type;
    void *   _value;
} any;
```

Since all **TypeCode** parameters have one of only three types (**string**, **TypeCode**, or **long**), the **_type** member will always be set to **TC_string**, **TC_TypeCode**, or **TC_long**, as appropriate. The **_value** member always points to the actual parameter datum. For example, the following code can be used to extract the name of a structure from a **TypeCode** of kind **tk_struct** in C:

```
#include <repostry.h> /* Interface Repository class */
#include <typedef.h>  /* Interface Repository TypeDef class */
#include <somtcnst.h> /* TypeCode constants */


TypeCode x;
Environment *ev = somGetGlobalEnvironment ();
TypeDef aTypeDefObj;
sequence(Contained) sc;
any parm;
string name;
Repository repo;

...

/* 1st, obtain a TypeCode from an Interface Repository object,
 * or use a TypeCode constant.
 */

repo = RepositoryNew ();
sc = _lookup_name (repo, ev,
    ”AttributeDescription”, -1, ”TypeDef”, TRUE);
if (sc._length) {
    aTypeDefObj = sc._buffer[0];
    x = __get_type (aTypeDefObj, ev);
    }
else
    x = TC_AttributeDescription;

if (TypeCode_kind (x, ev) == tk_struct) {
    parm = TypeCode_parameter (x, ev, 0); /* Get structure name */
    if (TypeCode_kind (parm._type, ev) != tk_string) {
        printf (”Error, unexpected TypeCode: ”);
        TypeCode_print (parm._type, ev);
    } else {
        name = *((string *)parm._value);
        printf (”The struct name is %s\n”, name);
    }
} else {
    printf (”TypeCode is not a tk_struct: ”);
    TypeCode_print (x, ev);
}
```

Related Information

Functions: TypeCodeNew, **TypeCode_alignment**, **TypeCode_copy**, **TypeCode_equal**, **TypeCode_free**, **TypeCode_kind**, **TypeCode_param_count**, **TypeCode_print**, **TypeCode_setAlignment**, **TypeCode_size**

TypeCode_print Function

Purpose

Writes all of the information contained in a given **TypeCode** to **stdout**.

IDL Syntax

void TypeCode_print ( );

Description

The **TypeCode_print** function can be used during program debugging to inspect the contents of a **TypeCode**. It prints (in a human-readable format) all of the information contained in the **TypeCode**. The format of the information shown by **TypeCode_print** is the same form that could be used by a C programmer to code the corresponding **TypeCodeNew** function call to create the **TypeCode**.

This function is a SOM-unique extension to the CORBA standard.

Parameters

tc	The **TypeCode** to be examined.
ev	A pointer to an **Environment** structure. The CORBA standard mandates the use of this structure as a standard way to return exception information when an error condition is detected.

Return Value

None. No exceptions are raised by this function.

Related Information

Functions: TypeCodeNew, **TypeCode_alignment**, **TypeCode_copy**, **TypeCode_equal**, **TypeCode_free**, **TypeCode_kind**, **TypeCode_param_count**, **TypeCode_parameter**, **TypeCode_setAlignment**, **TypeCode_size**

TypeCode_setAlignment Function

Purpose

Overrides the default alignment value associated with a given **TypeCode.**

IDL Syntax

void TypeCode_setAlignment (short *alignment***);**

Description

The **TypeCode_setAlignment** function overrides the default alignment value associated with a given **TypeCode.**

Parameters

tc	The **TypeCode** to receive the new alignment value.
ev	A pointer to an Environment structure.
alignment	A short integer that specifies the alignment value.

Related Information

Functions: TypeCodeNew, **TypeCode_alignment**, **TypeCode_equal**, **TypeCode_free**, **TypeCode_kind**, **TypeCode_param_count**, **TypeCode_parameter**, **TypeCode_print**, **TypeCode_size**

TypeCode_size Function

Purpose

Provides the minimum size of an instance of the abstract data type described by a given **TypeCode**.

IDL Syntax

long TypeCode_size ( );

Description

The **TypeCode_size** function is used to obtain the minimum size of an instance of the abstract data type described by a given **TypeCode**.

This function is a SOM-unique extension to the CORBA standard.

Parameters

tc	The **TypeCode** whose instance size is desired.
ev	A pointer to an **Environment** structure. The CORBA standard mandates the use of this structure as a standard way to return exception information when an error condition is detected.

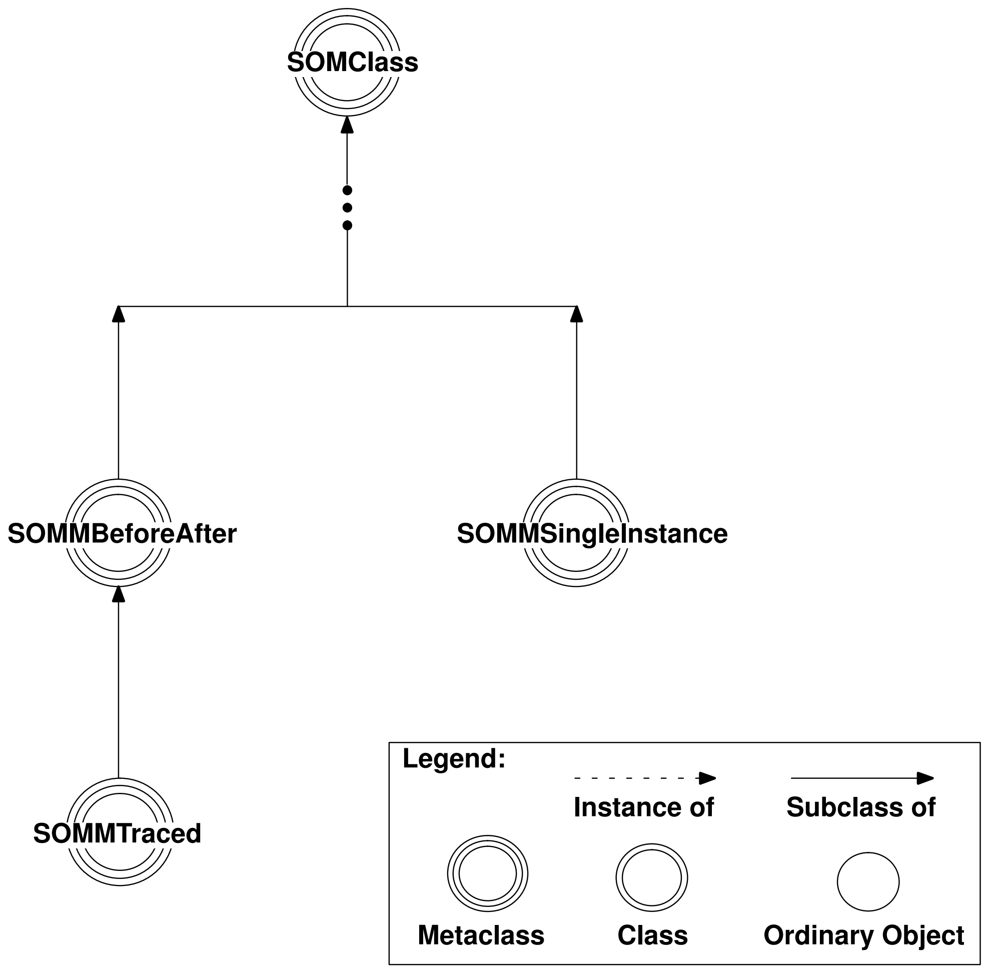
Return Value

The amount of memory needed to hold an instance of the data type described by a given **TypeCode**. No exceptions are raised by this function.

Related Information

Functions: TypeCodeNew, **TypeCode_alignment**, **TypeCode_copy**, **TypeCode_equal**, **TypeCode_free**, **TypeCode_kind**, **TypeCode_param_count**, **TypeCode_parameter**, **TypeCode_print**, **TypeCode_setAlignment**

Chapter 4. Metaclass Framework Reference



Metaclass Class Organization

SOMMBeforeAfter Metaclass

Description

SOMMBeforeAfter is a metaclass that defines two methods (**sommBeforeMethod** and **sommAfterMethod**), which are invoked before and after each invocation of every instance method. **SOMMBeforeAfter** is designed to be subclassed. Within the subclass, each of the two methods should be overridden with a method procedure appropriate to the particular application. The before and after methods are invoked on instances (ordinary objects) of a class whose metaclass is the subclass (or child) of **SOMMBeforeAfter**, whenever any method (*inherited* or *introduced*) of the class is invoked.

Warning: The **somDefaultInit** and **somFree** methods are among the methods that get before/after behavior. This implies that the following two obligations are imposed on the programmer of a **SOMMBeforeAfter** class. First, your implementation must guard against calling the **sommBeforeMethod** before **somDefaultInit** has executed, when the object is not yet fully initialized. Second, the implementation must guard against calling **sommAfterMethod** after **somFree**, at which time the object no longer exists.

SOMMBeforeAfter is thread-safe.

File Stem

sombacls

New Methods

None.

Overriding Methods

somDefaultInit

somInitMIClass

sommAfterMethod Method

Purpose

Specifies a method that is automatically called after execution of each client method.

IDL Syntax

void sommAfterMethod (
in SOMObject *object*,
in somId *methodID*,
**in void ****returnedvalue*,
in va_list *ap***);**

Description

The **sommAfterMethod** specifies a method that is automatically called after execution of each client method. The **sommAfterMethod** method is introduced in the **SOMMBeforeAfter** metaclass. The default implementation does nothing until it is overridden. The **sommAfterMethod** method is not called directly by the user. To define the desired "after" method, **sommAfterMethod** must be overridden in a metaclass that is a subclass (child) of the **SOMMBeforeAfter** metaclass.

Warning: somFree is among the methods that get before/after behavior, which implies that the following obligation is imposed on the programmer of a **sommAfterMethod**. Specifically, care must be taken to guard against **sommAfterMethod** being called after **somFree**, at which time the object no longer exists.

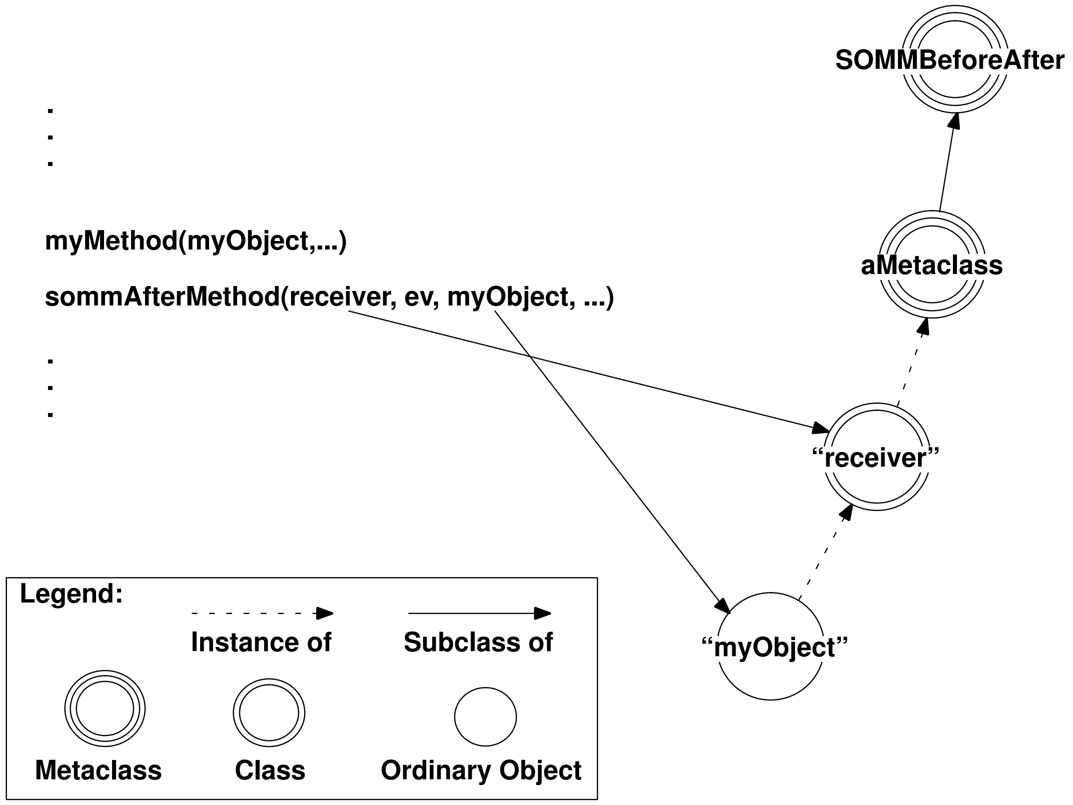
Parameters

Refer to the diagram in the following section for further clarification of these arguments.

receiver	A pointer to an object (class) of metaclass **SOMMBeforeAfter** representing the class object that supports the method (such as, "myMethod") for which the "after" method will apply.
ev	A pointer where the method can return exception information if an error is encountered. The dispatch method of **SOMMBeforeAfter** sets this parameter to NULL before dispatching the first **sommBeforeMethod**.
object	A pointer to the instance of the receiver on which the method is invoked.
methodId	The SOM ID of the method (such as, "myMethod") that was invoked.
returnedvalue	A pointer to the value returned by invoking the method ("myMethod") on an object.
ap	The list of input arguments to the method ("myMethod").

Example

The following figure shows an invocation of “myMethod” on “myObject”. Because “myObject” is an instance of a class whose metaclass is a subclass of **SOMMBeforeAfter**, “myMethod” is followed by an invocation of **sommAfterMethod** (note the user does not actually code the method). The adjacent figure illustrates the meaning of the parameters to **sommAfterMethod**.



An Example of Using sommAfterMethod

Original Class

SOMMBeforeAfter

Related Information

Methods: sommBeforeMethod

sommBeforeMethod Method

Purpose

Specifies a method that is automatically called before execution of each client method.

IDL Syntax

boolean sommBeforeMethod (
in SOMObject *object***,**
in somId *methodID***,**
in va_list *ap***);**

Description

The **sommBeforeMethod** specifies a method that is automatically called before execution of each client method. The **sommBeforeMethod** method is not called directly by the user. To define the desired "before" method, **sommBeforeMethod** must be overridden in a metaclass that is a subclass (child) of **SOMMBeforeAfter**. The default implementation does nothing until it is overridden.

Warning: somDefaultInit is among the methods that get before/after behavior, which implies that the following obligation is imposed on the programmer of a **sommBeforeMethod**. Specifically, care must be taken to guard against **sommBeforeMethod** being called before the **somDefaultInit** method has executed and the object is not yet fully initialized.

Parameters

Refer to the diagram in the following section for further clarification of these arguments.

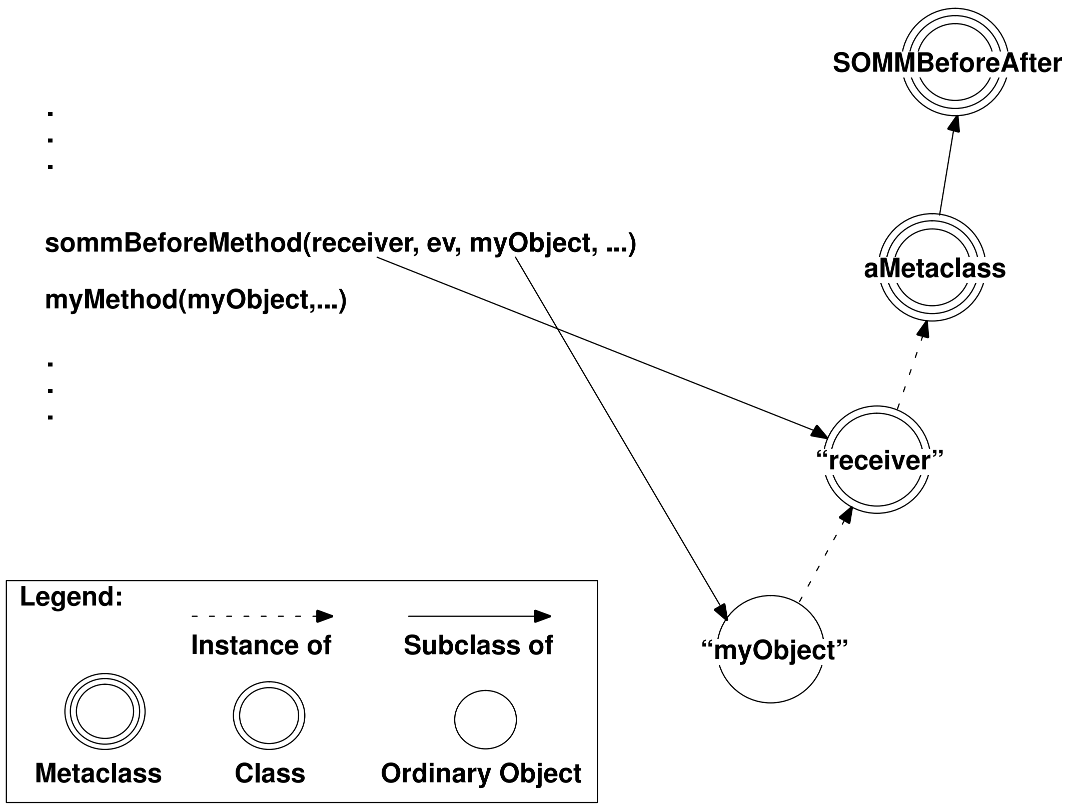
receiver	A pointer to an object (class) of metaclass **SOMMBeforeAfter** representing the class object that supports the method (such as, "myMethod") for which the "before" method will apply.
ev	A pointer where the method can return exception information if an error is encountered. The dispatch method of **SOMMBeforeAfter** sets this parameter to NULL before dispatching the first **sommBeforeMethod**.
object	A pointer to the instance of the *receiver* on which the method is invoked.
methodId	The SOM ID of the method (such as, "myMethod") that was invoked.
ap	The list of input arguments to the method ("myMethod").

Return Value

A **boolean** that indicates whether or not before/after dispatching should continue. If the value is TRUE, normal before/after dispatching continues. If the value is FALSE, the dispatching skips to the **sommAfterMethod** associated with the preceding **sommBeforeMethod**. This implies that the **sommBeforeMethod** must do any post-processing that might otherwise be done by the **sommAfterMethod**. Because before/after methods are paired within a **SOMMBeforeAfter** metaclass, this design eliminates the complexity of communicating to the **sommAfterMethod** that the **sommBeforeMethod** returned FALSE.

Example

The following figure shows an invocation of “myMethod” on “myObject”. Because “myObject” is an instance of a class whose metaclass is a subclass of **SOMMBeforeAfter**, “myMethod” is preceded by an invocation of **sommBeforeMethod** (note the user does not actually code the method). The adjacent figure illustrates the meaning of the parameters to **sommBeforeMethod.**



An Example of Using sommBeforeMethod

Original Class

SOMMBeforeAfter

Related Information

Methods: sommAfterMethod

SOMMSingleInstance Metaclass

Description

SOMMSingleInstance can be specified as the metaclass when a class implementor is defining a class for which only one instance can ever be created. The first call to <*className*>**New** in C, the **new** operator in C++, or the **somNew** method creates the one possible instance of the class. Thereafter, any subsequent “new” calls return the first (and only) instance.

Alternatively, the *method* **sommGetSingleInstance** can be used to accomplish the same purpose. The method offers an advantage in that the call site explicitly shows that something special is occurring and that a new object is not necessarily being created.

SOMMSingleInstance is thread-safe.

File Stem

snglicls

Base Class

SOMClass

Metaclass

SOMClass

Ancestor Classes

SOMClass

SOMObject

New Methods

sommGetSingleInstance

Overriding Methods

somInit

somNew

sommGetSingleInstance Method

Purpose

Gets the one instance of a specified class for which only a single instance can exist.

IDL Syntax

SOMObject sommGetSingleInstance ( );

Description

The **sommGetSingleInstance** method gets a pointer to the one instance of a class for which only a single instance can exist. A class can have only a single instance when its metaclass is the **SOMMSingleInstance** metaclass (or is a subclass of it).

The first call to <*className*>**New** in C, the **new** operator in C++, or the **somNew** method creates the one possible instance of the class. Thereafter, any subsequent "new" calls return the first (and only) instance. Using the **sommGetSingleInstance** method offers an advantage, however, in that the call site explicitly shows that something special is occurring and that a new object is not necessarily being created. (That is, the **sommGetSingleInstance** method creates the single instance if it does not already exist.)

Parameters

receiver	A pointer to a class object whose metaclass is **SOMMSingleInstance** (or is a subclass of it).
ev	A pointer where the method can return exception information if an error is encountered.

Return Value

The **sommGetSingleInstance** method returns a pointer to the single instance of the specified class.

Example

Suppose the class "XXX" is an instance of **SOMMSingleInstance**; then the following C code fragment passes the assertions.

```
x1  = XXXNew();
x2  = XXXNew();
assert( x1 == x2 );
x3 = _sommGetSingleInstance(  _somGetClass( x1 ), env );
assert( x2 == x3 );
```

Note that the method **sommGetSingleInstance** is invoked on the class object, because **sommGetSingleInstance** is a method introduced by the metaclass **SOMMSingleInstance**.

Original Class

SOMMSingleInstance

SOMMTraced Metaclass

Description

SOMMTraced is a metaclass that facilitates tracing of method invocations. Whenever a method (inherited or introduced) is invoked on an instance (simple object) of a class whose metaclass is **SOMMTraced**, a message prints to standard output giving the method parameters; then, after completion, a second message prints giving the returned value.

There is one more step for using **SOMMTraced**: nothing prints unless the environment variable SOMM_TRACED is set. If it is set to the empty string, *all* traced classes print. If the environment variable SOMM_TRACED is not the empty string, it should be set to the list of *names of classes* that should be traced. For example, for *csh* users, the following command turns on printing of the trace for “Collie” and “Chihuahua”, but not for any other traced class:

```
setenv SOMM_TRACED ”Collie Chihuahua”
```

SOMMTraced is thread-safe.

File Stem

somtrcls

Base Class

SOMMBeforeAfter

Ancestor Classes

SOMMBeforeAfter

SOMClass

SOMObject

Attributes

boolean sommTraceIsOn

This attribute indicates whether or not tracing is turned on for a class. This gives dynamic control over the trace facility.

New Methods

None.

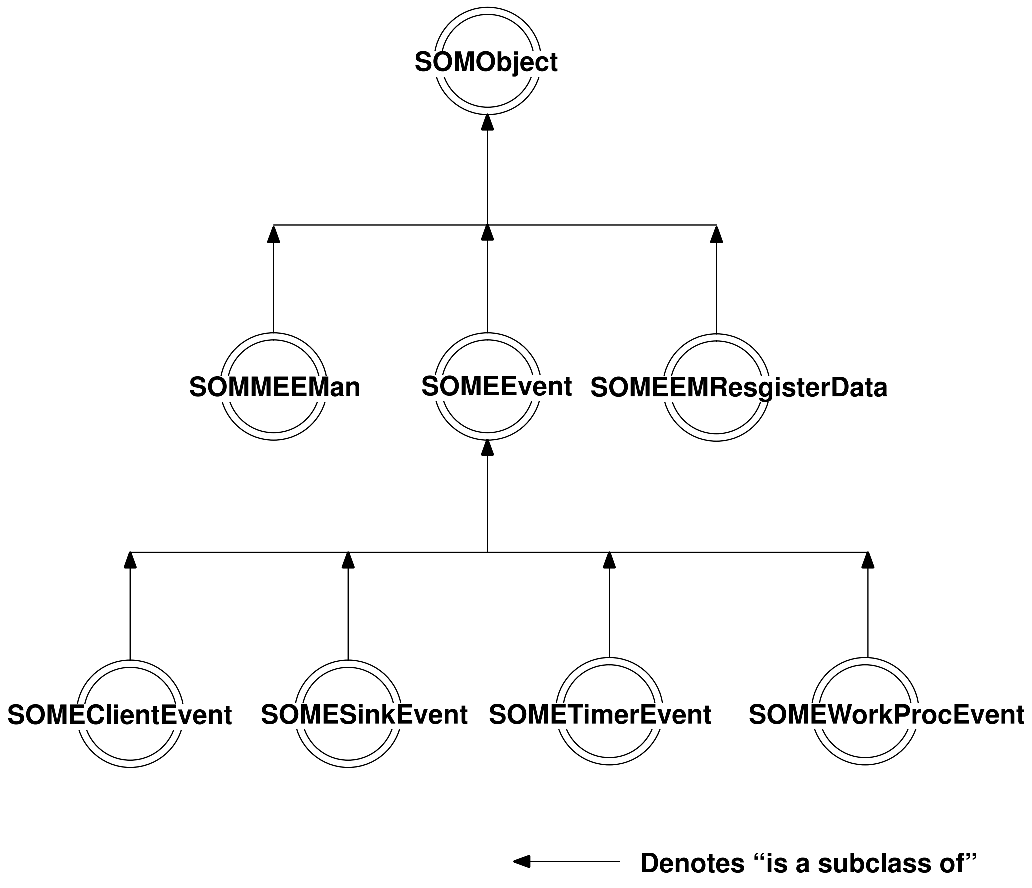
Overriding Methods

somInitMIClass

sommAfterMethod

sommBeforeMethod

Chapter 5. Event Management Framework Reference



Event Management Framework Class Organization

SOMEClientEvent Class

Description

This class describes generic client events within the Event Manager (EMan). Client Events are defined, created, processed and destroyed entirely by the application. The application can queue several types of client events with EMan. When a client event occurs, EMan passes an instance of this class to the callback routine. The callback can query this object about its type and obtain any event-specific information.

File Stem

clientev

Base

SOMEEvent

Metaclass

SOMClass

Ancestor Classes

SOMEEvent

SOMObject

New Methods

somevGetEventClientData

somevGetEventClientType

somevSetEventClientData

somevSetEventClientType

Overriding Methods

somInit

somevGetEventClientData Method

Purpose

Returns the user-defined data associated with a client event.

IDL Syntax

void* somevGetEventClientData ( );

Description

This method returns the user-defined data (if any) associated with the Client Event object. This associated data for a given client event type is passed to EMan at the time of registration.

Parameters

receiver	A pointer to an object of class **SOMEClientEvent**.
ev	A pointer to the **Environment** structure for the calling method.

Return Value

A pointer to user-defined client event data.

Original Class

SOMEClientEvent

Related Information

Methods: somevSetEventClientData

somevGetEventClientType Method

Purpose

Returns the type name of a client event.

IDL Syntax

string somevGetEventClientType ( );

Description

This method returns the client event type of the Client Event object. Client event type is a string name assigned to the event by the application at the time of registering the event.

Parameters

receiver	A pointer to an object of class **SOMEClientEvent**.
ev	A pointer to the **Environment** structure for the calling method.

Return Value

A null terminated string identifying the client event type.

Original Class

SOMEClientEvent

Related Information

Methods: somevSetEventClientType

somevSetEventClientData Method

Purpose

Sets the user-defined data of a client event.

IDL Syntax

```
void  somevSetEventClientData (
                                      in void* clientData);
```

Description

This method sets the user-defined event data (if any) of the Client Event object. This associated data for a given client event type is passed to EMan at the time of registration.

Parameters

receiver	A pointer to an object of class **SOMEClientEvent**.
ev	A pointer to the **Environment** structure for the calling method.
clientData	A pointer to user-defined data for this client event.

Original Class

SOMEClientEvent

Related Information

Methods: somevGetEventClientData

somevSetEventClientType Method

Purpose

Sets the type name of a client event.

IDL Syntax

```
void  somevSetEventClientType (
                                          in  string clientType);
```

Description

This method sets the client event type field of the Client Event object. Client event type is a string name assigned to the event by the application at the time of registering the event.

Parameters

receiver	A pointer to an object of class **SOMEClientEvent**.
ev	A pointer to the **Environment** structure for the calling method.
clientType	A null terminated character string identifying the client event type. The contents of this string are entirely up to the user. However, while using class libraries that also use client events one must make sure that there are no name collisions.

Original Class

SOMEClientEvent

Related Information

Methods: somevGetEventClientType

SOMEEMan Class

Description

The Event Manager class (EMan) is used to handle several input events. The main purpose of this class is to provide a service that can do a blocked (or timed) wait on several event sources concurrently. Typically, in a main program, one registers an interest in an event type with EMan and specifies a callback (a procedure or a method) to be invoked when the event of interest occurs. After all the necessary registrations are complete, the main program ends with a call to **someProcessEvents** in EMan. This call is non-returning. EMan then waits on all registered event sources. The application is completely event driven at this point (that is, it does something only when an event occurs). The control returns to EMan after processing each event. Further registrations can be done from within the callback routines. Unregistrations can also be done from within the callback routines.

For applications that want to have their own main loop, EMan provides a non-blocking call (the **someProcessEvent** method), which processes just one event (if any) and returns to the main loop immediately. Note that when this call is the only one in the application's main loop, CPU cycles are wasted in constantly polling for events. In this situation, the non-returning form of the **someProcessEvents** call is preferable.

AIX Specifics

On AIX this event manager supports Timer, Sink (any file, pipe, socket, or Message Queue), Client and WorkProc events.

OS/2 and Windows Specifics

On OS/2 and Windows, this event manager supports Timer, Sink (sockets only), Client, and WorkProc events.

Thread Safety

To cope with multi-threaded applications on OS/2, the event-manager methods are mutually exclusive (that is, at any time only one thread can be executing inside of EMan). If an application thread needs to stop EMan from running (that is, to achieve mutual exclusion with EMan), it can use the two methods **someGetEManSem** and **someReleaseEManSem** to acquire and release EMan semaphores. On AIX or Windows, since threads are not supported (at present), calling these two methods has no effect.

File Stem

eman

Base Class

SOMObject

Metaclass

SOMMSingleInstance

Ancestor Classes

SOMObject

New Methods

someGetEManSem

someChangeRegData

someProcessEvent

someProcessEvents

someQueueEvent

someRegister

someRegisterEv

someRegisterProc

someReleaseEManSem

someShutdown

someUnRegister

Overriding Methods

somInit

somUninit

someChangeRegData Method

Purpose

Changes the registration data associated with a specified registration ID.

IDL Syntax

void someChangeRegData (
in long *registrationId*,
in SOMEEMRegisterData *registerData***);**

Description

This method is called to change the registration data associated with an existing registration of EMan. The existing registration is identified by the *registrationId* parameter. This ID must be the one returned by EMan when the event interest was originally registered with EMan. Further, the registration must be active (that is, it must not have been unregistered). The result of providing a non-existent or invalid registration ID is a “no op”.

Parameters

receiver	A pointer to an object of class **SOMEEMan**.
ev	A pointer to the **Environment** structure for the calling method.
registrationId	The registration ID of the event interest whose data is being changed.
registerData	A pointer to the registration data object whose contents will replace the existing registration information with EMan.

Example

```
#include <eman.h>
SOMEEMan *EManPtr;
SOMEEMRegisterData *data;
Environment *Ev;
long RegId;

  ...
_someChangeRegData(EManPtr, Ev, RegId, data);
```

Original Class

SOMEEMan

Related Information

Methods: someRegister, **someRegisterEv**, **someRegisterProc**

someGetEManSem Method

Purpose

Acquires EMan semaphores to achieve mutual exclusion with EMan's activity.

IDL Syntax

void someGetEManSem ( );

Description

When EMan is used on OS/2, multiple threads can invoke methods on EMan concurrently. EMan protects its internal data by acquiring SOM toolkit semaphores. The same semaphores are made available to users of EMan through the methods **someGetEManSem** and **someReleaseEManSem**. If an application desires to prevent EMan event processing from interfering with its own activity (in another thread, of course), then it can call the **someGetEManSem** method and acquire EMan semaphores. EMan activity will resume when the application thread releases the same semaphores by calling **someReleaseEManSem**.

Callers should not hold this semaphore for too long, since it essentially stops EMan activity for that duration and may cause EMan to miss some important event processing. The maximum duration for which one can hold this semaphore depends on how frequently EMan must process events.

On AIX or Windows, calling this method has no effect.

Parameters

receiver	A pointer to an object of class **SOMEEMan**.
ev	A pointer to the **Environment** structure for the calling method.

Example

```
#include <eman.h>
SOMEEMan *EManPtr;
Environment *Ev;

  ...
_someGetEManSem(EManPtr, Ev);
   /* Do the work that needs mutual exclusion with EMan */
_someReleaseEManSem(EManPtr, Ev);
```

Original Class

SOMEEMan

Related Information

Methods: someReleaseEManSem

someProcessEvent Method

Purpose

Processes one event.

IDL Syntax

void someProcessEvent (
in unsigned long *mask***);**

Description

Processes one event. This call is non-blocking. If there are no events to process it returns immediately. The mask specifies which events to process. The mask is formed by OR'ing the bit constants specified in the **eventmsk.h** file.

Parameters

receiver	A pointer to an object of class **SOMEEMan**.
ev	A pointer to the **Environment** structure for the calling method.
mask	A bit mask indicating the types of events to look for and process.

Example

```
#include <eman.h>

main()
{
Environment *testEnv = somGetGlobalEnvironment();
SOMEEMan *some_gEMan = SOMEEManNew();
 /* Do some registrations */
...
while (1) {
        _someProcessEvent(some_gEMan,  testEnv,
                                EMProcessTimerEvent |
                                EMProcessSinkEvent |
                                EMProcessClientEvent );
 /***  Do other main loop work, if needed. ***/
}
} /* end of main */
```

Original Class

SOMEEMan

Related Information

Methods: someProcessEvents, **someRegister**, **someRegisterEv**, **someRegisterProc**

someProcessEvents Method

Purpose

Processes infinite events.

IDL Syntax

void someProcessEvents ( );

Description

This call loops forever waiting for events and dispatching them. The only way this can be broken is by calling **someShutdown** in a callback routine. It is a programming error to call this method without having registered interest in any events with EMan. Typically, a call to this method is the last statement in the main program of an application.

Parameters

receiver	A pointer to an object of class **SOMEEMan**.
ev	A pointer to the **Environment** structure for the calling method.

Example

```
#include <eman.h>

main()
{
Environment *testEnv = somGetGlobalEnvironment();
SOMEEMan *some_gEMan = SOMEEManNew();
 /* Do some registrations */
...
_someProcessEvents(some_gEMan,  testEnv);
} /* end of main */
```

Original Class

SOMEEMan

Related Information

Methods: someProcessEvent, **someRegister**, **someRegisterEv**, **someRegisterProc**

someQueueEvent Method

Purpose

Enqueues the specified client event.

IDL Syntax

void someQueueEvent (
in SOMEClientEvent *event***);**

Description

Client events are defined, created, processed and destroyed by the application. EMan simply provides a means to enqueue and dequeue client events. Client events can be used in several ways. For example, if an application component wants to handle an input message arriving on a socket at a later time than when it arrives, it can receive the message in the socket callback routine, create a client event out of it, and queue it with EMan. EMan can be asked for the client event at a later time when the application is ready to handle it. Client events can also be useful to hide the origin of event sources (that is, the original event handlers receive the events and create client events in their place).

Dequeue is not a user-visible operation. Once a client event is queued, only EMan can dequeue it.

Parameters

receiver	A pointer to an object of class **SOMEEMan**.
ev	A pointer to the **Environment** structure for the calling method.
event	A pointer to the **SOMEClientEvent** object.

Example

```
#include <eman.h>
SOMEClientEvent *clientEvent1;

clientEvent1 = SOMEClientEventNew();
/* create a client event of type ”ClientType1” */
_somevSetEventClientType( clientEvent1, testEnv, ”ClientType1” );
_somevSetEventClientData( clientEvent1, testEnv, ”Test Msg”);
   ...

/* whenever it is desired to cause this client event to happen,
call someQueueEvent Method with this clientEvent */
_someQueueEvent(some_gEMan, env, clientEvent1);
```

Original Class

SOMEEMan

someRegister Method

Purpose

Registers an object/method pair with EMan, given a specified *registerData* object.

IDL Syntax

long someRegister (
in SOMEEMRegisterData *registerData*,
in SOMObject *targetObject*,
in string *targetMethod*,
in void **targetData*);

Description

This method allows for registering an event of interest with EMan, with an object method as the callback. It is assumed that the target method has been declared as using OIDL callstyle. The event of interest and its details are filled in a registration data object *registerData*. The information about the callback routine is indicated by *targetObject* and *targetMethod*.

A mismatch between the target method's callstyle and the registration method used (that is, **someRegister** vs. **someRegisterEv**) can result in unpredictable results.

Note: The target method is called using name-lookup method resolution.

Parameters

receiver	A pointer to an object of class **SOMEEMan**.
ev	A pointer to the **Environment** structure for the calling method.
registerData	A pointer to the registration data object that contains all the necessary information about the event for which an interest is being registered with EMan.
targetObject	A pointer to the object that is the target of the callback method.
targetMethod	The name of the callback method.
targetData	A pointer to a data structure to be passed to the callback method when the event occurs.

Return Value

The registration ID.

Example

```
#include <eman.h>
#include <emobj.h>

Environment *testEnv = somGetGlobalEnvironment();
some_gEMan = SOMEEManNew();/* create an EMan object */
data = SOMEEMRegisterDataNew( ); /* create a reg data object */
target = EMObjectNew(); /* create a target object */

/* reRegister a timer event */
_someClearRegData( data, env );
_someSetRegDataEventMask( data, env, EMTimerEvent, NULL );
_someSetRegDataTimerInterval( data, env, 100 );
regId1 = _someRegister( some_gEMan, env, data, target,
                        ”eventMethod”, ”Timer 100” );
```

Original Class

SOMEEMan

Related Information

Methods: someRegisterEv, **someRegisterProc**, **someUnRegister**

Also see the **callstyle** modifier of the SOM Interface Definition Language described in Chapter 4, “Implementing SOM Classes” of the *SOM Toolkit User’s Guide*.

someRegisterEv Method

Purpose

Registers the (object, method, **Environment** parameter) combination of a callback with EMan, given a specified *registerData* object.

IDL Syntax

```
long  someRegisterEv (
                           in SOMEEMRegisterData registerData,
                           in SOMObject targetObject,
                           inout Environment  callbackEv,
                           in string targetMethod,
                           in void *targetData);
```

Description

This method allows for registering an event interest with EMan with an object method as callback. The *callbackEv* is used as the environment pointer when EMan makes the callback. It is assumed that the target method has been declared as using IDL callstyle. The event of interest and its details are filled in a registration data object *registerData*. The information about the callback routine is indicated by *targetObject* and *targetMethod*.

A mismatch in the target method's callstyle and the registration method called (**someRegister** vs. **someRegisterEv**) can result in unpredictable results.

Note: The target method is called using name-lookup method resolution.

Parameters

receiver	A pointer to an object of class **SOMEEMan**.
ev	A pointer to the **Environment** structure for the calling method.
registerData	A pointer to registration data object that contains all the necessary information about the event for which an interest is being registered with EMan.
targetObject	A pointer to the object which is the target of the callback method
callbackEv	A pointer to the Environment structure to be passed to the callback method
targetMethod	The name of the callback method.
targetData	A pointer to a data structure to be passed to the callback method when the event occurs.

Return Value

The registration ID.

Example

```
#include <eman.h>
#include <emobj.h>

Environment *testEnv = somGetGlobalEnvironment();
Environment *targetEv = somGetGlobalEnvironment();
some_gEMan = SOMEEManNew();/* create an EMan object */
data = SOMEEMRegisterDataNew( ); /* create a reg data object */
target = EMObjectNew(); /* create a target object */

/* reRegister a timer event */
_someClearRegData( data, env );
_someSetRegDataEventMask( data, env, EMTimerEvent, NULL );
_someSetRegDataTimerInterval( data, env, 100 );
regId1 = _someRegisterEv( some_gEMan,env, data, target,targetEv,
                        "eventMethod", "Timer 100" );
 /* eventMethod of target is assumed to use callstyle=idl */
```

Original Class

SOMEEMan

Related Information

Methods: someRegister, **someRegisterProc**, **someUnRegister**

Also see the **callstyle** modifier in the SOM Interface Definition Language described in Chapter 4, “Implementing SOM Classes,” in the *SOM Toolkit User’s Guide.*

someRegisterProc Method

Purpose

Register the procedure with EMan given the specified *registerData*.

IDL Syntax

long someRegisterProc (
in SOMEEMRegisterData *registerData,*
in EMRegProc **targetProcedure,*
in void **targetData***);**

Description

The **someRegisterProc** method allows for registering an event of interest with EMan, with a specified procedure as the callback. The event of interest and its details are provided through a registration data object *registerData*. The information about the callback procedure is indicated by *targetProcedure*.

Parameters

receiver	A pointer to an object of class **SOMEEMan**.
ev	A pointer to the **Environment** structure for the calling method.
registerData	A pointer to registration data object that contains all the necessary information about the event for which an interest is being registered with EMan.
targetProcedure	A pointer to the procedure (callback) that is called when the registered event occurs.
targetData	A pointer to a data structure to be passed to the callback procedure when the event occurs.

Return Value

The registration ID.

Example

```
#include <eman.h>

void MyCallBack(SOMEEvent *event, void *somedata){
 ...
}

Environment *testEnv = somGetGlobalEnvironment();
some_gEMan = SOMEEManNew();/* create an EMan object */
data = SOMEEMRegisterDataNew( ); /* create a reg data object */

/* reRegister a timer event */
_someClearRegData( data, env );
_someSetRegDataEventMask( data, env, EMTimerEvent, NULL );
_someSetRegDataTimerInterval( data, env, 100 );
regId1 = _someRegisterProc( some_gEMan, env, data,
                        MyCallBack, ”Timer 100” );
```

Original Class

SOMEEMan

Related Information

Methods: someRegister, **someRegisterEv**, **someUnRegister**

someReleaseEManSem Method

Purpose

Releases the semaphore obtained by the **someGetEManSem** method.

IDL Syntax

void someReleaseEManSem ( );

Description

When EMan is used on OS/2, multiple threads can invoke methods on EMan concurrently. EMan protects its internal data by acquiring SOM toolkit semaphores. The same semaphores are made available to users of EMan through the methods **someGetEManSem** and **someReleaseEManSem**. If an application desires to prevent EMan's event processing from interfering with its own activity (in another thread, of course), then it can call the **someGetEManSem** method and acquire EMan semaphores. EMan activity will resume when the application thread releases the same semaphores by calling **someReleaseEManSem**.

Callers should not hold this semaphore for too long, since it essentially stops EMan activity for that duration and may cause EMan to miss some important event processing. The maximum duration for which one can hold this semaphore depends on how frequently EMan must process events.

On AIX or Windows, calling this method has no effect.

Parameters

receiver	A pointer to an object of class **SOMEEMan**.
ev	A pointer to the **Environment** structure for the calling method.

Example

```
#include <eman.h>
SOMEEMan *EManPtr;
Environment *Ev;

  ...
_someGetEManSem(EManPtr, Ev);
  /* Do the work that needs mutual exclusion with EMan */
_someReleaseEManSem(EManPtr, Ev);
```

Original Class

SOMEEMan

Related Information

Methods: someGetEManSem

someShutdown Method

Purpose

Shuts down an EMan event loop. (That is, this makes the **someProcessEvents** return!)

IDL Syntax

void someShutdown ( );

Description

This can be called from a callback routine to break the someProcessEvents loop.

Parameters

receiver	A pointer to an object of class **SOMEEMan**.
ev	A pointer to the **Environment** structure for the calling method.

Example

```
#include <eman.h>
SOMEEMan *some_gEMan;

void MyCallBack(SOMEEvent *event, void *somedata){
 ...
 _someShutdown(some_gEMan, env);
}
main()
{
Environment *testEnv = somGetGlobalEnvironment();
SOMEEMan *some_gEMan = SOMEEManNew();
 /* Do some registrations. At least one involving MyCallBack */
...
_someProcessEvents(some_gEMan,  testEnv);
}
```

Original Class

SOMEEMan

Related Information

Methods: someProcessEvents

someUnRegister Method

Purpose

Unregisters the event interest associated with a specified *registrationId* within EMan.

IDL Syntax

void someUnRegister (
in long *registrationId***);**

Description

When an application is no longer interested in a given event, it can unregister the event interest from EMan. EMan will stop making callbacks on this event, even if the event source continues to be active and generates events.

Parameters

receiver	A pointer to an object of class **SOMEEMan**.
ev	A pointer to the **Environment** structure for the calling method.
registrationId	The registration ID of the event that needs to be unregistered.

Example

```
#include <eman.h>
long regId1;

 ...
/* Register a timer */
regId1 = _someRegisterEv( some_gEMan,env, data, target,targetEv,
                        ”eventMethod”, ”Timer 100” );
 ....
/* Unregister the timer */
_someUnRegister(some_gEMan, env, regId1);
```

Original Class

SOMEEMan

Related Information

Methods: someRegister, **someRegisterEv**, **someRegisterProc**

SOMEEMRegisterData Class

Description

This class is used for holding registration information for event types to be registered with EMan. EMan extracts all needed information from this object and saves the information in its internal data structures. An instance of this class must be created, properly initialized, and passed to the registration methods of EMan for registering interest in any kind of event.

File Stem

emregdat

Base

SOMObject

Metaclass

SOMClass

Ancestor Classes

SOMObject

New Methods

someClearRegData

someSetRegDataClientType

someSetRegDataEventMask

someSetRegDataSink

someSetRegDataSinkMask

someSetRegDataTimerCount

someSetRegDataTimerInterval

Overriding Methods

somInit

somUninit

someClearRegData Method

Purpose

Clears the registration data.

IDL Syntax

void someClearRegData ( );

Description

This method initializes all fields of a RegData object to their default values.

Parameters

receiver	A pointer to an object of class **SOMEEMRegisterData**.
ev	A pointer to the **Environment** structure for the calling method.

Original Class

SOMEEMRegisterData

someSetRegDataClientType Method

Purpose

Sets the type name for a client event.

IDL Syntax

void someSetRegDataClientType (
in string *clientType***);**

Description

Client events are defined, created, processed, and destroyed entirely by the application. The application can queue several types of client events with EMan. This method sets the client event type field of the registration data object. Thus, this information is communicated to EMan, helping it deal with enqueueing and dequeing the different client events.

Parameters

receiver	A pointer to an object of class **SOMEEMRegisterData**.
ev	A pointer to the **Environment** structure for the calling method.
clientType	A null-terminated character string identifying the client event type. The contents of this string are entirely up to the user. However, while using class libraries that also use client events, one must make sure that there are no name collisions.

Original Class

SOMEEMRegisterData

Related Information

Methods: someClearRegData

someSetRegDataEventMask Method

Purpose

Sets the generic event mask within the registration data using NULL terminated event type list.

IDL Syntax

void someSetRegDataEventMask (
in long *eventType***,**
in va_list *ap***);**

Description

This allows setting the event mask within the registration data object. Essentially, this tells EMan what kind of event is being registered with it. The event type list is a series of constants defined in the **eventmsk.h** file. Although the current interface supports a NULL terminated list of event types, currently each registration with EMan names only one event type. Thus, one usually gives only one named constant as the event type and follows it with a NULL parameter (see the following example).

Parameters

receiver	A pointer to an object of class **SOMEEMRegisterData**.
ev	A pointer to the **Environment** structure for the calling method.
eventType	A bit constant indicating the type of event being registered with EMan.
ap	Additional event types (usually NULL).

Example

```
#include <eman.h>
long regId1;
int msgsock;

 ...
/* Register msgsock socket with EMan for further communication */
_someClearRegData( data, env );
_someSetRegDataEventMask( data, env, EMSinkEvent, NULL );
/* The above call enables EMan to know (during registration) that
we are talking about a Sink Event */
_someSetRegDataSink( data, env, msgsock );
_someSetRegDataSinkMask( data, env, EMInputReadMask);

regId = _someRegisterProc( some_gEMan, env, data,
                        ReadSocketAndPrint, ”READMSG” );
```

Original Class

SOMEEMRegisterData

Related Information

Methods: someSetRegDataSink, **someClearRegData**

someSetRegDataSink Method

Purpose

Sets the file descriptor (or socket ID, or message queue ID) for the sink event.

IDL Syntax

void someSetRegDataSink (
in long *sink***);**

Description

This method enables setting the true type of an event object. Typically, a subclass of Event calls this method (or overrides this method) to set the event type to indicate its true class(type).

Parameters

receiver	A pointer to an object of class **SOMEEMRegisterData**.
ev	A pointer to the **Environment** structure for the calling method.
sink	An integer value indicating the file descriptor for input/output. It can also be a socket ID, pipe ID or a message queue ID.

Original Class

SOMEEMRegisterData

Related Information

Methods: someClearRegData

someSetRegDataSinkMask Method

Purpose

Sets the sink mask within the registration data object.

IDL Syntax

void someSetRegDataSinkMask (
in unsigned long *sinkmask***);**

Description

The sink mask within the registration data allows one to express interest in different events of the same event source. For example, using this mask one can express interest in being notified when there is input for reading, when the resource is ready for writing output, or just when exceptions occur.

Parameters

receiver	A pointer to an object of class **SOMEEMRegisterData**.
ev	A pointer to the **Environment** structure for the calling method.
sinkmask	A bit mask indicating the types of events of interest on a given sink.

Example

```
#include <eman.h>
long regId1;
int msgsock;

 ...
/* Register msgsock socket with EMan for further communication */
_someClearRegData( data, env );
_someSetRegDataEventMask( data, env, EMSinkEvent, NULL );
_someSetRegDataSink( data, env, msgsock );
_someSetRegDataSinkMask( data, env,
                         EMInputReadMask|EMInputExceptMask);
/* The above call expresses interest in knowing when there is
 input to be read from the socket and when there is an exception
condition associated with this socket. */
regId = _someRegisterProc( some_gEMan, env, data,
                        ReadSocketAndPrint, ”READMSG” );
```

Original Class

SOMEEMRegisterData

Related Information

Methods: someSetRegDataSink, **someClearRegData**

someSetRegDataTimerCount Method

Purpose

Sets the number of times the timer will trigger, within the registration data.

IDL Syntax

void someSetRegDataTimerCount (
in long *count***);**

Description

The **someSetRegDataTimerCount** method sets the number of times the timer will trigger, within the registration data. The default behavior is for the timer to trigger indefinitely.

Parameters

receiver	A pointer to an object of class **SOMEEMRegisterData**.
ev	A pointer to the **Environment** structure for the calling method.
count	An integer indicating the number of times the timer event has to occur.

Example

```
#include <eman.h>
long regId1;

 ...
/* Register a timer */
_someClearRegData( data, env );
_someSetRegDataEventMask( data, env, EMTimerEvent, NULL );
_someSetRegDataTimerInterval( data, env, 100 );
_someSetRegDataTimerCount(data, env, 1);
/* make this a one time timer event */
regId1 = _someRegister( some_gEMan,env, data, target,
                        ”eventMethod”, ”Timer 100” );
```

Original Class

SOMEEMRegisterData

Related Information

Methods: someClearRegData

someSetRegDataTimerInterval Method

Purpose

Sets the timer interval within the registration data.

IDL Syntax

void someSetRegDataTimerInterval (
in long *interval***);**

Description

This call allows setting the timer interval (in milliseconds) within the registration data object.

Parameters

receiver	A pointer to an object of class **SOMEEMRegisterData**.
ev	A pointer to the **Environment** structure for the calling method.
interval	An integer indicating the timer interval in milliseconds.

Example

```
#include <eman.h>
long regId1;

 ...
/* Register a timer */
_someClearRegData( data, env );
_someSetRegDataEventMask( data, env, EMTimerEvent, NULL );
_someSetRegDataTimerInterval( data, env, 100 );
/* Sets the timer interval to 100 milliseconds */
regId1 = _someRegister( some_gEMan,env, data, target,
                        ”eventMethod”, ”Timer 100” );
```

Original Class

SOMEEMRegisterData

Related Information

Methods: someClearRegData

SOMEEvent Class

Description

This is the base class for all generic events within the Event Manager (EMan). It simply timestamps an event before it is passed to a callback routine. The event type is set to the true type by a subclass. The types currently used by the Event Management Framework are defined in the **eventmsk.h** file. Any subclass of this class must avoid name and value collisions with the **eventmsk.h** file.

File Stem

event

Base

SOMObject

Metaclass

SOMClass

Ancestor Classes

SOMObject

New Methods

somevGetEventTime

somevGetEventType

somevSetEventTime

somevSetEventType

Overriding Methods

somInit

somevGetEventTime Method

Purpose

Returns the time of the generic event in milliseconds.

IDL Syntax

unsigned long somevGetEventTime ( );

Description

EMan timestamps every event before dispatching it. The current time is obtained from the operating system (for example, using a **gettimeofday** call), is converted to milliseconds, and is given as the value of the timestamp. When this function is called, the event timestamp is returned.

Parameters

receiver	A pointer to an object of class **SOMEEvent**.
ev	A pointer to the **Environment** structure for the calling method.

Return Value

An event timestamp in milliseconds.

Original Class

SOMEEvent

Related Information

Methods: somevSetEventTime

somevGetEventType Method

Purpose

Returns the type of the generic event.

IDL Syntax

unsigned longsomevGetEventType ( );

Description

This method returns the true type of a given event object (for example, to identify the particular subclass of the event object). The type is an integer valued constant defined in the **eventmsk.h** file.

Parameters

receiver	A pointer to an object of class **SOMEEvent**.
ev	A pointer to the **Environment** structure for the calling method.

Return Value

A type value (an integer constant defined in the **eventmsk.h** file).

Original Class

SOMEEvent

Related Information

Methods: somevSetEventType

somevSetEventTime Method

Purpose

Sets the time of the generic event (time is in milliseconds).

IDL Syntax

```
void  somevSetEventTime (
                              in unsigned long time);
```

Description

EMan timestamps every event before dispatching it. The current time is obtained from the operating system (for example, using a **gettimeofday** call), converted to milliseconds, and is given as the value of the timestamp. When an event occurs, EMan sets the timestamp of the event by calling this method.

Parameters

receiver	A pointer to an object of class **SOMEEvent**.
ev	A pointer to the **Environment** structure for the calling method.
time	The time of day expressed in milliseconds.

Original Class

SOMEEvent

Related Information

Methods: somevGetEventTime

somevSetEventType Method

Purpose

Sets the type of the generic event.

IDL Syntax

void somevSetEventType (
in unsigned long *type***);**

Description

This method enables setting the true type of an event object. Typically, a subclass of **SOMEEvent** calls this method (or overrides this method) to set the event type to indicate its true type.

Parameters

receiver	A pointer to an object of class **SOMEEvent**.
ev	A pointer to the **Environment** structure for the calling method.
type	An integer value indicating the type of the event (a constant defined in the **eventmsk.h** file).

Original Class

SOMEEvent

Related Information

Methods: somevGetEventType

SOMESinkEvent Class

Description

This class describes a sink event that is generated by EMan when it notices activity on a registered sink. On AIX, a sink refers to any file descriptor (file open for reading or writing), any pipe descriptor, a socket ID or a message queue ID. On OS/2 or Windows, a sink refers to a socket ID. One can register for three types of interest in a sink: Read interest, Write interest, and Exception interest. (See the **eventmsk.h** file to determine the appropriate bit constants and see the **someSetRegDataSinkMask** method for their use.)

EMan passes an instance of this class as a parameter to the callback registered for Sink Events. The callback can query the instance for some information on the sink.

File Stem

sinkev

Base

SOMEEvent

Metaclass

SOMClass

Ancestor Classes

SOMEEvent

SOMObject

New Methods

somevGetEventSink

somevSetEventSink

Overriding Methods

somInit

somevGetEventSink Method

Purpose

Returns the sink, or source of I/O, of the generic sink event.

IDL Syntax

long somevGetEventSink ( );

Description

The sink ID in the SinkEvent is returned. For message queues it is the queue ID, for files it is the file descriptor, for sockets it is the socket ID, and for pipes it is the pipe descriptor.

Parameters

receiver	A pointer to an object of class **SOMESinkEvent**.
ev	A pointer to the **Environment** structure for the calling method.

Return Value

An integer value indicating the file descriptor for input/output. It can also be a socket ID, pipe ID or a message queue ID.

Original Class

SOMESinkEvent

Related Information

Methods: somevSetEventSink

somevSetEventSink Method

Purpose

Sets the sink, or source of I/O, of the generic sink event.

IDL Syntax

void somevSetEventSink (
in long *sink***);**

Description

The sink ID in the SinkEvent is set. For message queues, it is the queue ID; for files it is the file descriptor; for sockets it is the socket ID; and for pipes it is the pipe descriptor.

Parameters

receiver	A pointer to an object of class **SOMESinkEvent**.
ev	A pointer to the **Environment** structure for the calling method.
sink	An integer value indicating the file descriptor for input/output. It can also be a socket ID, pipe ID, or a message queue ID.

Original Class

SOMESinkEvent

Related Information

Methods: somevGetEventSink

SOMETimerEvent Class

Description

This class describes a timer event that is generated by EMan when any of its registered timers pops.

EMan passes an instance of this class as a parameter to the callbacks registered for Timer Events. The callback can query the instance for information on the timer interval and on any generic event properties.

File Stem

timerev

Base

SOMEEvent

Metaclass

SOMClass

Ancestor Classes

SOMEEvent

SOMObject

New Methods

somevGetEventInterval

somevSetEventInterval

Overriding Methods

somInit

somevGetEventInterval Method

Purpose

Returns the interval of the generic timer event (time in milliseconds).

IDL Syntax

void somevGetEventInterval ( );

Description

The **somevGetEventInterval** method returns the interval of the generic timer event (time in milliseconds).

Parameters

receiver	A pointer to an object of class **SOMETimerEvent**.
ev	A pointer to the **Environment** structure for the calling method.

Return Value

The interval time in milliseconds.

Original Class

SOMETimerEvent

Related Information

Methods: somevSetEventInterval

somevSetEventInterval Method

Purpose

Sets the interval of the generic timer event (in milliseconds).

IDL Syntax

```
void  somevSetEventInterval (
                              in  long interval);
```

Description

The **somevSetEventInterval** method sets the interval of the generic timer event (in milliseconds).

Parameters

receiver	A pointer to an object of class **SOMETimerEvent**.
ev	A pointer to the **Environment** structure for the calling method.
interval	The timer interval in milliseconds.

Original Class

SOMETimerEvent

Related Information

Methods: somevGetEventInterval

SOMEWorkProcEvent Class

Description

This class describes a work procedure event object. It currently has no methods of its own. However, it sets the event type in its super class to say “EMWorkProcEvent” to help identify itself. These events are created and dispatched by EMan when a work procedure (something that the application wants to run when no other events are happening) is registered with EMan.

EMan passes an instance of this class as a parameter to the callback registered for WorkProc Events.

File Stem

workprev

Base

SOMEEvent

Metaclass

SOMClass

Ancestor Classes

SOMEEvent

SOMObject

New Methods

None.

Overriding Methods

SOMClass

somInit

Index

Vos remarques sur ce document / Technical publication remark form

Titre / Title : Bull DPX/20 SOMobjects Base Toolkit Programmer's Reference Manual

Nº Reférence / Reference Nº : 86 A2 28AQ 01

Daté / Dated : June 1995

ERREURS DETECTEES / ERRORS IN PUBLICATION

AMELIORATIONS SUGGEREES / SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Vos remarques et suggestions seront examinées attentivement
Si vous désirez une réponse écrite, veuillez indiquer ci-après votre adresse postale complète.

Your comments will be promptly investigated by qualified technical personnel and action will be taken as required.
If you require a written reply, please furnish your complete mailing address below.

NOM / NAME : ______________________ Date : ________

SOCIETE / COMPANY : ______________________

ADRESSE / ADDRESS : ______________________

____

Remettez cet imprimé à un responsable BULL ou envoyez-le directement à :

Please give this technical publication remark form to your BULL representative or mail to:

BULL S.A. CEDOC
Atelier de Reproduction
FRAN–231
331 Avenue Patton BP 428
49005 ANGERS CEDEX
FRANCE

PLACE BAR CODE IN LOWER
LEFT CORNER

BULL S.A. CEDOC
Atelier de Reproduction
FRAN–231
331 Avenue Patton BP 428
49005 ANGERS CEDEX
FRANCE

ORDER REFERENCE
86 A2 28AQ 01

Bull

Utiliser les marques de découpe pour obtenir les étiquettes.
Use the cut marks to get the labels.

DPX/20
AIX
SOMobjects Base
Toolkit
Programmer's
Reference Manual
86 A2 28AQ 01

DPX/20
AIX
SOMobjects Base
Toolkit
Programmer's
Reference Manual
86 A2 28AQ 01

DPX/20
AIX
SOMobjects Base
Toolkit
Programmer's
Reference Manual
86 A2 28AQ 01