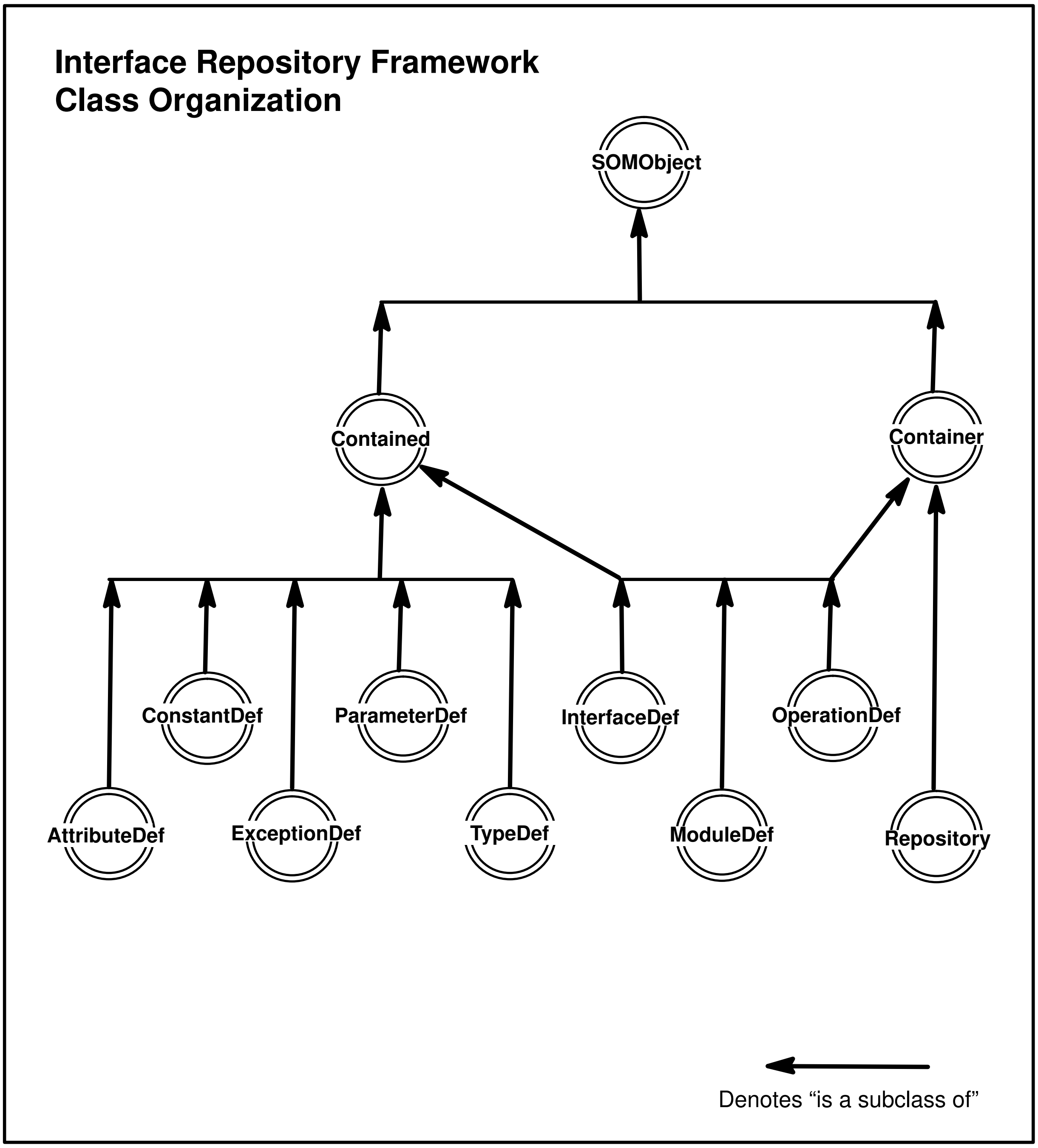
Interface Repository Framework Reference



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

Listed below is 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. 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 (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

Listed below is 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. 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**.

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.

Listed below is 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

within
describe

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);
```

Caution: 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
lookup_name
describe_contents

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 valid list above) 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: lookup_name, describe_contents

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 valid list above) 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"* 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 valid list above) or "all", to 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

Listed below is 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. 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

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.

Listed below is each available attribute, with its corresponding type in parentheses, followed by a description of its purpose:

base_interfaces (sequence<RepositoryId>)
The sequence of **RepositoryId**s 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
within
describe

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

Listed below is 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. 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 (OperationMode)
The **OperationMode** of the operation (method), either NORMAL or ONEWAY.

contexts (sequence<ContextIdentifier>)

The list of **ContextIdentifier**s 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

Listed below is 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

Container, 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.

Return Value

None.

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

Listed below is 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_kind, TypeCode_param_count, TypeCode_parameter, TypeCode_setAlignment, TypeCode_size, TypeCode_free, TypeCode_print

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_kind, TypeCode_param_count, TypeCode_parameter, TypeCode_size, TypeCode_free, TypeCode_print, TypeCode_setAlignment

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_kind, TypeCode_param_count, TypeCode_parameter, TypeCode_copy, TypeCode_free, 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_equal, TypeCode_kind, TypeCode_param_count, TypeCode_parameter, TypeCode_size, TypeCode_copy, TypeCode_print, TypeCode_setAlignment

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.

Table 1 (see following page) 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**.

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 above. No exceptions are raised by this function.

Related Information

Functions: TypeCodeNew, TypeCode_alignment, TypeCode_equal, TypeCode_param_count, TypeCode_parameter, TypeCode_copy, TypeCode_free, TypeCode_print, TypeCode_setAlignment, TypeCode_size

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 repeat for each member ———	
		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 foreign type.
		string	The implementation context.
		long	The size of an instance.

Table 1. **TypeCode** information per **TCKind** category.

†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. See the section entitled "Using the tk_foreign TypeCode" in Chapter 7 of the *SOM Toolkit User's Guide* for more information about using foreign **TypeCode**s in SOM IDL files.

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.

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 "Syntax" section above. 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 s**truct** 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_parameter, TypeCode_print, TypeCode_size, TypeCode_setAlignment

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_equal, TypeCode_kind, TypeCode_parameter, TypeCode_copy, TypeCode_free, TypeCode_print, TypeCode_size, TypeCode_setAlignment

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_equal, TypeCode_kind, TypeCode_param_count, TypeCode_copy, TypeCode_free, TypeCode_print, TypeCode_size, TypeCode_setAlignment**

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_equal, TypeCode_kind, TypeCode_param_count, TypeCode_parameter, TypeCode_copy, TypeCode_free, TypeCode_size, TypeCode_setAlignment

TypeCode_setAlignment Function

Purpose

Sets the alignment value for a given **TypeCode.**

IDL Syntax

void TypeCode_setAlignment (short *alignment***);**

Description

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.

Return Value

None.

Related Information

Functions: TypeCodeNew, TypeCode_alignment, TypeCode_equal, TypeCode_kind, TypeCode_param_count, TypeCode_parameter, TypeCode_size, TypeCode_free, TypeCode_print

TypeCode_size Function

Purpose

Provides the 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 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.

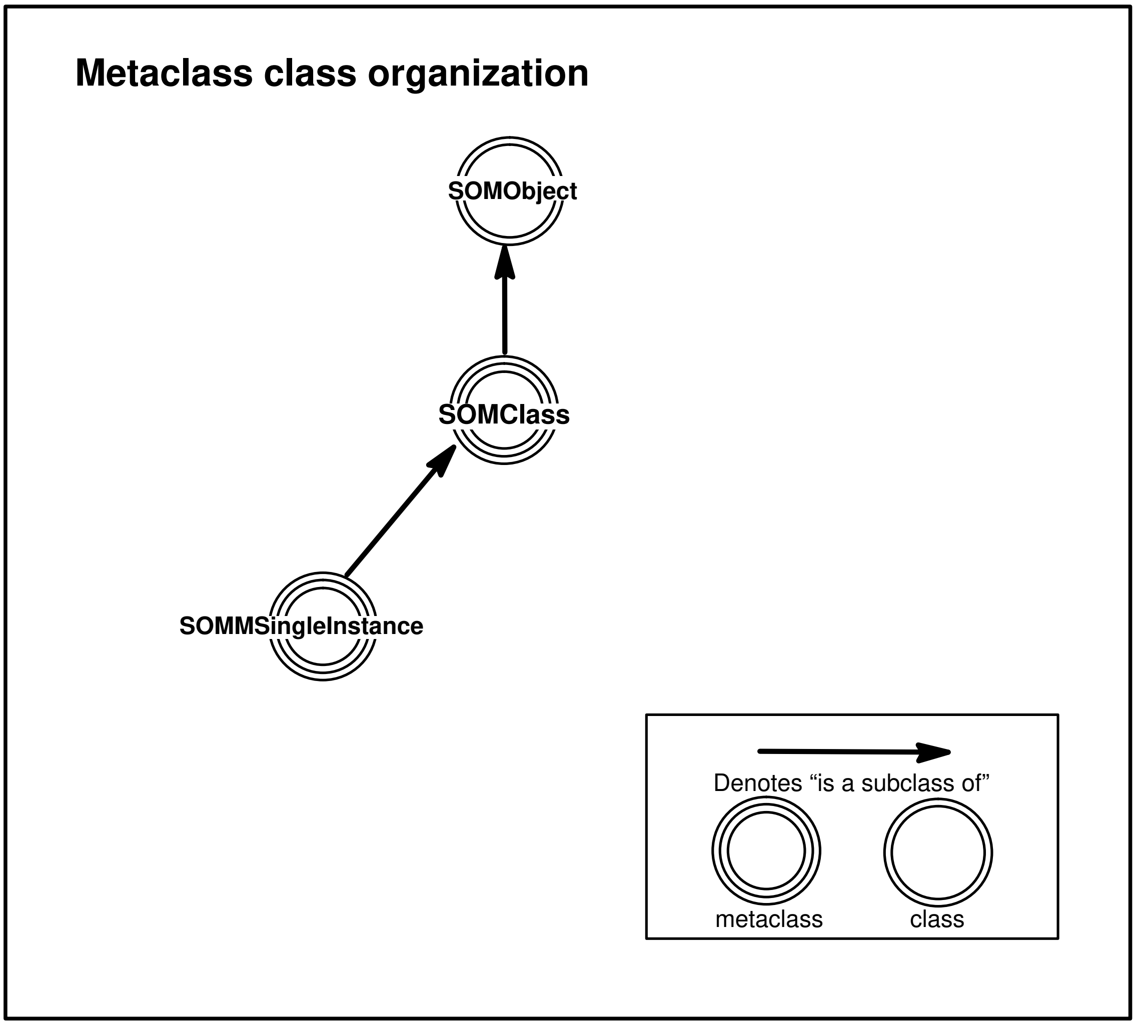
Return value

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_equal, TypeCode_kind, TypeCode_param_count, TypeCode_parameter, TypeCode_copy, TypeCode_free, TypeCode_print, TypeCode_setAlignment

Utility Metaclass and Methods Reference



SOMMSingleInstance Class

Description

SOMMSingleInstance is a metaclass provided with the SOM Toolkit. It can be specified as the metaclass when 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.

File Stem

snglicls

Base Classes

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, however, 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. (That is, the **sommGetSingleInstance** method creates the single instance if it does not already exist.)

Parameters

receiver	A pointer to an object (class) 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.

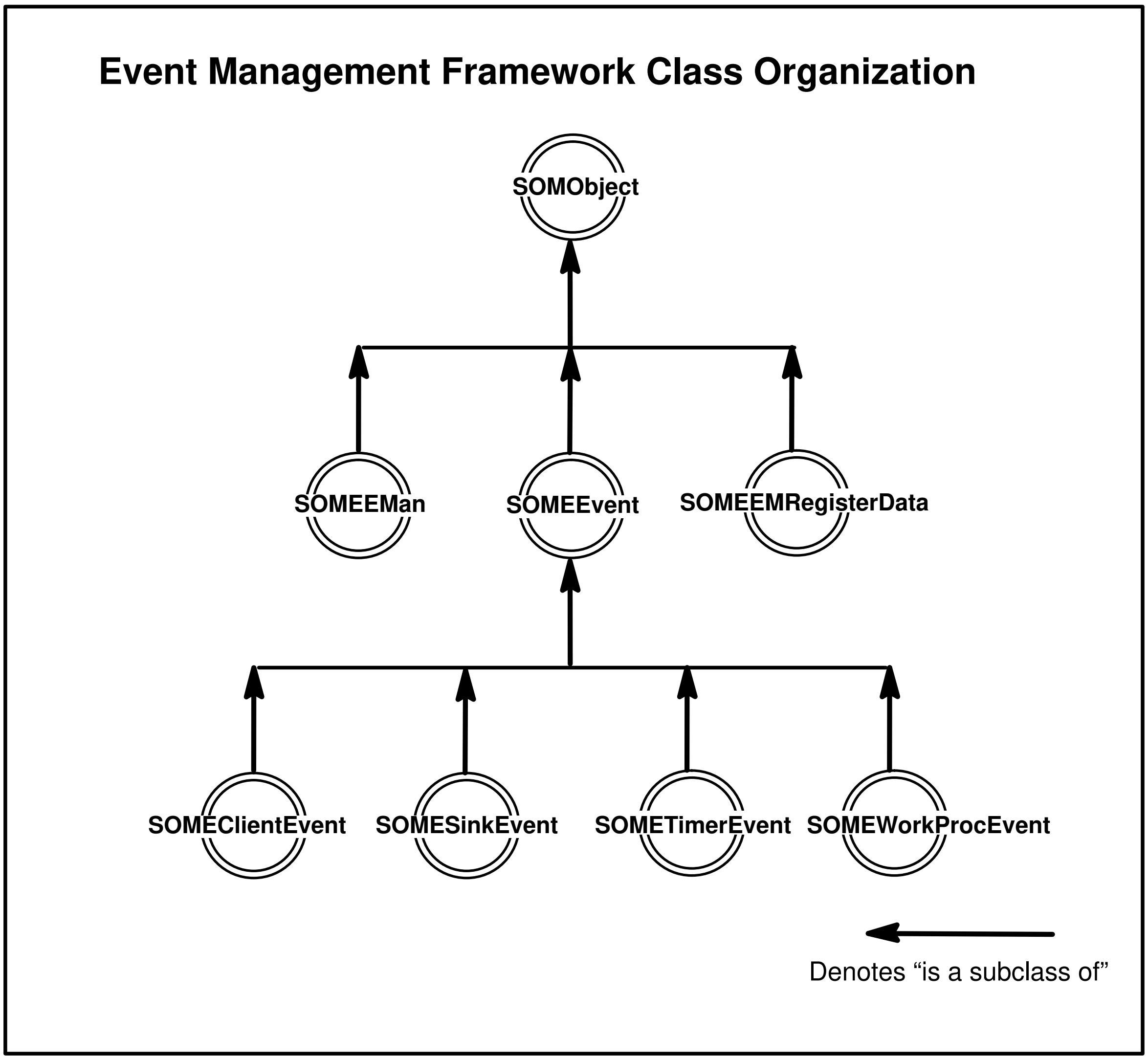
```
x1  = XXXNew();
x2  = XXXNew();
assert( x1 == x2 );
x3 = _sommGetSingleInstance(  _somGetClass( x1 ), env );
assert( x2 == x3 );
```

Note that the method **sommGetSingleInstance** is invoked on the class object, because **sommGetSingleInstance** is a method introduced by the metaclass **SOMMSingleInstance.**

Original Class

SOMMSingleInstance

Event Management Framework Reference



SOMEClientEvent Class

Description

This class describes generic client events within the Event Manager. 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.

Return Value

None.

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.

Return Value

None.

Original Class

SOMEClientEvent

Related Information

Methods: somevGetEventClientType

SOMEEMan Class

Description

The Event Manager class (EMan for short) 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.

<u>AIX Specifics</u>:
On AIX this event manager supports Timer, Sink (any file, pipe, socket, or Message Queue), Client and WorkProc events.

<u>OS/2 Specifics</u>:
On OS/2 this event manager supports Timer, Sink(sockets only), Client, and WorkProc events.

<u>Thread Safety</u>:
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 semaphore(s). On AIX, since AIX does not support threads (at present), calling these two methods has no effect.

File Stem

eman

Base Class

SOMObject

Metaclass

SOMMSingleInstance

Ancestor Classes

SOMObject

New Methods

someGetEManSem
someReleaseEManSem
someChangeRegData
someProcessEvent
someProcessEvents

someQueueEvent
someRegister
someRegisterEv
someRegisterProc
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.

Return Value

None.

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 semaphore(s) 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 semaphore(s). The same semaphore(s) 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 semaphore(s). EMan activity will resume when the application thread releases the same semaphore(s) 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, 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.

Return Value

None.

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 "eventmsk.h".

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.

Return Value

None.

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, someRegisterProc, someRegisterEv

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 an application's main program.

Parameters

receiver	A pointer to an object of class **SOMEEMan.**
ev	A pointer to the **Environment** structure for the calling method.

Return Value

None.

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, someRegisterProc, someRegisterEv

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 **clientevent** object.

Return Value

None.

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

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 semaphore(s). The same semaphore(s) 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 semaphore(s). EMan activity will resume when the application thread releases the same semaphore(s) 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, 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.

Return Value

None.

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.

Return Value

None.

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.

Return Value

None.

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.

Return Value

None.

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.

Return Value

None.

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 eventmsk.h. 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 example below).

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).

Return Value

None.

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.

Return Value

None.

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.

Return Value

None.

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.

Return Value

None.

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.

Return Value

None.

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. 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 eventmsk.h. Any subclass of this class must avoid name and value collisions with eventmsk.h.

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 long somevGetEventType ( );

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 eventmsk.h.

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 eventmsk.h).

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.

Return Value

None.

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 eventmsk.h).

Return Value

None.

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, 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 eventmsk.h file to determine the appropriate bit constants and see method **someSetRegDataSinkMask** 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.

Return Value

None.

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.

Return Value

None.

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

somInit

Index

E

F

G

I

T

U

W