SOS reference manual for prototype V4

Groupe Sor

▶ **To cite this version:**

Groupe Sor. SOS reference manual for prototype V4. RT-0108, INRIA. 1989, pp.107. inria-00070058

HAL Id: inria-00070058
https://inria.hal.science/inria-00070058

Submitted on 19 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P 105
78153 Le Chesnay Cedex
France
Tél. (1) 39 63 55 11

Rapports Techniques

N° 108

Programme 3

SOS REFERENCE MANUAL FOR PROTOTYPE V4

The SOR group

Juin 1989

RT_0108

SOS Reference Manual for Prototype V4

Manuel de référence pour le prototype SOS V4

The SOR group
Institut National de la Recherche en Informatique et Automatique
BP 105, 78153 Le Chesnay Cédex, France
e-mail: sor@sor.inria.fr

February 1989

Abstract

This is the SOS Reference Manual. It is available in this printed form, or on line via the Unix command SOSMAN[1]. This Manual documents all the SOS concepts and primitives in detail. It complements the *SOS Programmer's Manual* (available separately) which contains a step-by-step information on how to program in SOS.

Résumé

Ceci est le manuel de référence pour SOS V4. Il est disponible sous la forme imprimée ici présente, ou par la commande Unix SOSMAN[1]. Ce manuel comprend la documentation détaillée des concepts et des primitives de SOS. Il est complété par le *Manuel du Programmeur SOS*, qui apprend, par l'exemple et pas à pas, comment programmer une application SOS. Le Manuel du Progreammeur est imprimé séparément.

NAME

intro – introduction to SOS

DESCRIPTION

An SOS prototype is composed of a kernel program, **sos**, predefined contexts for acquaintance service, **ASmain**, storage service, **SSmain**, name service, **NSmain** and communication service, **CSmain**, and whatever contexts you chose to run. A context is simply a specially-prepared Unix process (see *sosCC(1)*). Contexts may be run either automatically by naming them in the **predefContexts** file of the current directory, or by starting them (once **sos** and **predefined contexts** have finished initializing) from Unix, e.g. from the shell. Within a context multiple tasks my be run, using the C++ task primitives. Communication between contexts is possible if the requestor or *client* imports a *proxy* of the requested *service* (see *proxy(2)*, *dynamic(2)*). Proxies communicate with their principal with the **crossInvoke** kernel primitive (see *crossInvoke(2)*, *setTrapRef(2)*).

SOS programs must be prepared under Unix. The following preparation utilities are available: sosCC the C++ compiler (modified to allow dynamic linking and object migration, and with automatic linking to the appropriate libraries).

There is also a short description of the use of the dynamic classes, in page *dynamic(2)*. Page *sosman(1)* explains the use of the SOS manual reader.

LIMITATIONS

This programmer's manual is for the SOS Prototype V4, running on a Sun-3. It is a prototype version intended to test the basic SOS ideas and run SOS programs. With respect to a fully functional SOS, it has the following limitations:

(1) runs on top of Unix, not the bare machine

(2) Inadequate protection of group membership, and of access to objects in general.

(3) No inter-object dependencies.

SEE ALSO

sosCC(1), sosman(1), crossInvoke(2), dynamic(2), proxy(2), setTrapRef(2), sos(7), Asmain(7), CSmain(7), NSmain(7), SSmain(7)

NAME

CC – C++ translator (with SOS extensions).

SYNOPSIS

CC [*option...*] *file...*

DESCRIPTION

CC (capital CC) translates C++ source code to C source code. The command uses *cpp*(1) for preprocessing, *cfront* and *cfront.dl* for syntax and type checking, and *cc*(1) or *gcc*(l) for code generation.

CC takes arguments ending in

.c to be C++ source programs; they are compiled, and each object program is left on the file whose name is that of the source with **.o** substituted for **.c**.

.s to be assembly source programs; they are assembled, producing **.o** files.

CC interprets the following options (the two first are SOS extensions):

–DL Run dynamic link cfront *(cfront.dl)*

+z Optimizes dynamic methods strings. String tables are not generated, so an error message can't list the method name at run-time.

–C Prevent *cpp* and *cfront* from removing comments.

–E Run only *cpp* on the **.c** files and send the result to standard output.

–F Run only *cpp* and *cfront* on the .c files, and send the result to standard output.

–Fc Like the **–F** option, but the output is C source code suitable as a **.c** file for *cc(1)* or *gcc*(l).

–*.suffix* Instead of using standard output for the **–E , –F** or **–Fc** options, place the output from each .c file on a file with the corresponding *.suffix*.

+V Accept regular C function declarations; use the */usr/include* directory **#include** files. Support for this option is not guaranteed in future releases.

+L Generate source line number information using the format "#line %d" instead of "# %d".

+x *file* Read a file of sizes and alignments. Each line contains three fields: a type name, the size (in bytes), and the alignment (in bytes). This option is useful for cross compilations.

+S Spy on *cfront*; that is, print some information on *stderr*.

See *ld*(1) for loader options, *as*(1) for assembler options, *cc*(1) for code generation options, and *cpp*(1) for preprocessor options (in the Unix manual).

FILES

file.c	input file
file..c	cfront output
file.o	object file
a.out	linked output
/lib/cpp	C preprocessor
cfront	C front end
cfront.dl	dynamic link C front end
/bin/cc	C compiler
/lib/libc.a	standard C library; see Section (3) in the *UNIX System V Programmer Reference Manual*
/lib/libC.a	C++ library
libD.a	dynamic linker library
/usr/include/CC	standard directory for **#include** files
/usr/include	standard directory for **#include** files when the +V option is used

SEE ALSO

sosCC(1), mergexport(1), exception(1), dynamic(1), gcc(l)
in Unix manual: cc(1), monitor(3), prof(1), ld(1)
Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley 1986.
B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall 1978.
"On the use of the dynamic linker in SOS V4", in the directory doc/dlink of the current distribution.

DIAGNOSTICS

The diagnostics produced by *CC* itself are intended to be self-explanatory. Occasional messages may be produced by the assembler or loader. No messages should be produced by *cc*(1) or *gcc*(1).

BUGS

Some "used before set" warnings are wrong.

There is a (temporary) hole in the C++ type system allowing C++ programs to use C libraries. When a name is overloaded the first function of that name (only) can be linked to a library compiled by *cc* or *gcc*. Thus, the declaration

```
overload read(int,char*,int), read(vector*);
```

will allow the system call *read*(2) to be used together with user defined functions of the same name. Use of this facility may lead to unexpected behavior. For example, had the other **read()** been declared first, or had the system **read()** not been declared, then the user's **read()** would have been called by library functions like *scanf*(3).

Dynamic link errors (and internal bugs) begin with the message "Error dynamic link".

On Sun Release 3.[2-4], it is not possible to compile in one pass. First compile individual with the **-c** option, then link them together with the **-o** option.

NAME

dllib – Dynamic Linking LIBrary

SYNOPSIS

dllib [*option*] *file.[ao]*

DESCRIPTION

Force linkage of libraries.
If method definitions of an exported module call library functions (like *printf* of the standard library *libc.a)*, this function should be linked with the importer code.

Dllib is provided to build object files from libraries, files which must be statically linked with importer module.

Dllib uses output of *nm(1)*, scanned in an executable named *dlpp* and provided in the libD/DLPP directory of the SOS distribution.

The default output file is *dllib.o*

OPTIONS

File can be a library file *file.a* or an object file *file.o*.
Options are interpreted as follows :

–o *outfile*
Name the output file *outfile*.

–f *file[.o]*
file is ignored by *dllib* if is included in library argument *file.a*

–d *datum*
Datum (global variable) is ignored by *dllib* if included in object file (of the library argument) *file.[ao]*.

–t *text* *Text* is ignored by *dllib* if included in object file (of the library argument) *file.[ao]*

EXAMPLE

% dllib -o dlibc.o -f on_exit /lib/libc.a
generates an object file *dlibc.o* from the standard library */lib/libc.a*, suppressing strange features in the dynamic linker introduced by the file *on_exit.o*.

% dllib -o dtermcap.o /lib/termcap
% sosCC import.c dtermcap.o -ltermcap
allows an imported module to call procedures within the library *termcap*.

SEE ALSO

nm(1).
"On the use of the dynamic linker in SOS V4", in the directory doc/dlink of the current distribution.

BUGS

Options *-f*, *-d*, *-t* are provided to by-pass strange features. Use of shared libraries should be a good thing.

NAME

mergexport – compile C++ programs for SOS

SYNOPSIS

mergexport [*option*] *file...*

DESCRIPTION

Compiles and links C++ files with the necessary options to create SOS exportable file.
Mergexport is provided to group in one binary exportable file, separated compilation of proxy files. The dynamic class declaration syntax (see *dynamic (2))* accepts a list of files but in the current version, he sole first file is performed. A proxy can be splitted in several source files, but separated compilation must be followed by call to *mergexport* for the importation purpose.
The default output file is the first file name with suffix .o

OPTIONS

File can be *.o* or *.c* files.
Options are interpreted as follows :

–o *outfile*
Rename the outfile (similar to the same *cc* option).

–E, –F, –Fc
These options are passed verbatim to sosCC (see *CC(1)).*

+Z Verbose option. Print all the compilation steps.

EXAMPLE

% mergexport f1.c f2.c
generates a binary file named f1.o. To rename this file, use:
% mergexport -o output f1.c f2.c

SEE ALSO

CC(1), sosCC(1), dynamic(2), gcc(l)

BUGS

When using default output option, the produced file can override the first binary file (if any), as:
% mergexport f1.o f2.c
This command provide an output binary file f1.o, separated compilation of previous f1.o and f2.o. In this case, use –o option.

NAME

sosCC – compile C++ programs for SOS

SYNOPSIS

sosCC [*option*] *file...*

DESCRIPTION

Compiles C++ files with the necessary options and libraries, to create SOS executables.

OPTIONS

Options are passed verbatim to *CC*. With no option or the *–o contextName* option, *sosCC* makes a binary for an SOS context.

SEE ALSO

CC(1), mergexport(1), exception(1), dynamic(2), gcc(l)

NAME

sosman – read SOS manual pages.

SYNOPSIS

man [*section*] *title*

DESCRIPTION

sosman is a simple shell script which uses the Unix command **man(1)** to read the SOS manual pages from directory ~**sos/v4/man.**

The SOS manual is divided into 8 sections:

1 describes Unix utilities for SOS.

2 describes the kernel and basic classes.

3 describes the Acquaintance Service.

4 describes the Storage Service.

5 describes the Name Service.

6 describes the Communication Service.

7 describes the SOS utilities.

8 describes the maintenance commands for SOS.

SEE ALSO

In Unix manual: man(1), man(5)

NAME

segmentDesc::allocate – create a segment and assign it to the segment descriptor

segmentDesc::free – destroy the segment referred by the segment descriptor

DECLARATION (in segment.h)

```
void
segmentDesc::allocate (long size =0);
void
segmentDesc::free ();
```

SYNOPSIS

```
#include <sos/sos.h>
...
segmentDesc seg ( . . . );
seg.allocate (1000);
seg.free ();
```

DESCRIPTION

segmentDesc::allocate creates a segment of size *size* or, if *size* is 0 or omitted, of size **maxSize** (see **segmentDesc(2)**). The size of the segment can not be greater than **maxSize.**

segementDesc::free deallocates the segment (i.e., frees its memory) referred by the segment descriptor. Only the descriptor which was used to *allocate* the segment may be used to *free* it.

EXCEPTIONS

sgNotAllowed

operation is not in capability list (see **segmentDesc(2)**); or attempt to deallocate a segment which was not allocated using this segment descriptor.

sgBadSize

the **size** argument is greater than **maxSize** (see *segmentDesc(2)*), or both **size** and **maxSize** are 0 or omitted.

sgOutOfMemory

not enough memory to allocate the segment

SEE ALSO

segmentDesc(2)

NAME

segmentDesc::assign – assign a (sub)segment of an existing (sub)segment to the segment descriptor

DECLARATION (In sos.h)

```
void
segmentDesc::assign (byte* address, long size =0);
void
segmentDesc::assign (segmentDesc* baseDesc, long offset =0, long size =0);
void
```

DESCRIPTION

The first form assigns a subsegment with the start address equal to *address*, and size equal to the *size* argument; or, if *size* is 0 or omitted, to *maxSize* (see *segmnetDesc(2)).*

The second form assigns a subsegment with a start address equal to the start address of the (sub)segment referred by *baseDesc*, plus the *offset* argument. The size of the subsegment is equal to the *size* argument, or, if *size* is 0 or omitted, to the size of the *baseDesc* (sub)segment minus *offset.*

The size of the subsegment can not be greater than *maxSize* (see *segmentDesc(2)).*

EXCEPTIONS

sgNotLocal

operation is invoked for a remote segment

sgBadSize

the size of assigned subsegment is 0 or greater than *maxSize* (see *segmentDesc(2));* or (in the second form) the bounds of the requested subsegment are not included within the bounds of the original segment.

sgNotASegment

baseDesc does not refer to a valid segment

SEE ALSO

segmentDesc(2)

NAME

code – intermediate object for code of migrated object

DECLARATION (in code.h)

```
#include "sos.h"
/* typedef int (**vtab)(); defined in sos.h */

dynamic class code : public sosObject{
  ...
 public:
  code (char* className, char* fileName ...) raises(noCode);
  code (char* className, ref* baseRef, char* fileName...)
                raises(noCode);
  ~code();

  int       classNameLength();
  void      getClassName( char* buf );
  int       classDtblLength();
  void      getDtbl( vtab tfunc );

  int       howmanyFiles();
  int       fileNameLength( int index = 0 );
  void      getFileName( char* buf, int index = 0 );
};
```

SYNOPSIS

```
/* in exporter, principal.c */
#include "sos.h"

char* className = "proxyClass";
char* fileName = "proxyClass.o";
ref*  baseRef;
 ...
code * my_code = new code (className, fileName, 0);

/* or */

code * my_code = new code (className, baseRef, fileName, 0);
```

ARGUMENTS

The *className* argument is the name of the class, which code is to be migrated.

The *fileName* argument is the name of file which contains the code. This file has been previously created by the *sosCC* command (see *sosCC(1)*). The file name is searched in the Unix directories named in the environnement variable DLPATH. The code for a class may be in several files, all arguments insered between *fileName* and the final zero are also considered as names for code files.

The *n* argument is the range of the file name for the code file. It is 0 by default, i.e. the range of the *fileName* argument.

The *baseRef* argument is the reference of the code object for the class from which the class *className* is derived. If the reference of the base code for a derived class is not given, at importation time, it will be searched from the default name "/export/baseClassName.o" with the Name Service.

DESCRIPTION

One can create a code object at run-time, each time it is needed, or create it once and to store it within the Storage Service, using the SOS utility makecode (see *makecode(7)).*

In order to export proxies, a provider should first create a *code* object for the proxy's code (or retrieve its reference from the nameService if the *code* object has been stored). Within the execution of *giveProxy*, the principal uses *setCodeRef* to set the code object reference.

The *code* object is automatically migrated into the client context if and when needed, using the normal migration mechanisms of the Acquaintance Service.

The *classNameLength* method returns the string length of the class name (without the null-ended character).

The *getClassName* method fills the *buf* argument with the class name.

The *classDtblLength* method returns the index number of the dynamic method table attached to the code object.

The *getDtbl* method fills the *tfunc* argument with addresses of the dynamic method table attached to the code object.

The *howmanyFiles* method returns the files number attached to the code object.

The *fileNameLength* method returns the string length of the file argument *index*. Default index is 0.

The *getFileName* method fills the *buf* argument of the *index* argument. Default index is 0.

EXAMPLE

The access method describes above are provided to supply *copies* information of the code object. A typical use is:

```
int length = code->classNameLength();
if( length ){
   char* name = new char[ length+1 ];
   code->getClassName( name );
   ...
}
```

EXCEPTIONS

Contructors of class code raise *noCode* when dynamic load and link of code has failed.

SEE ALSO

makecode(1), giveProxy(2), setCodeRef(2)

NAME

segmentDesc::copyTo – copy a (sub)segment into an other.

DECLARATION (in sos.h)

```
void
segmentDesc::copyTo (segmentDesc* targetDesc);
```

DESCRIPTION

Copy the contents of the (sub)segment referred by the segment descriptor into the (sub)segment referred by the *targetDesc*. The target (sub)segment must be already allocated and sufficiently large.

EXCEPTIONS

sgNotASegment

targetDesc descriptor or the source segment does not refer to a valid (sub)segment

sgBadSize

the size of the source (sub)segment is greater than the size of the target (sub)segment.

sgNotAllowed

the source descriptor does not have the **sgCopyFrom** capability, or the target descriptor does not have the **sgCopyTo** capability.

SEE ALSO

segmentDesc(2)

NAME

countLock– Lock class for objects synchronization

DECLARATION (in countLock.h)

```
class countLock{
   ...
 public:
   countLock(int n)      ;
   ~countLock();
   void wait();
   void set();
   void unset();
   int isLocked();
};
```

SYNOPSIS

```
#include <sos/countLock.h>

countLock lock;
   ...
   lock.P();
   ...
   if (lock.isLocked)
      lock.V();
   ...
```

DESCRIPTION

A *countLock* object allows to acquire a great number of locks (limited by the max positive value of an integer).

Its contructor allows to acquire a number of *n* locks.

The *wait* procedure puts the current task in a pending state until all locks are released. MAX_WAITS tasks may wait for a countLock object to become unlocked.

The *set* procedure allows to acquire one lock more.

The *unset* procedure allows to release one lock. If there are no more locks, all pending tasks are resumed.

The *isLocked* procedure returns non-zero if the countLock is locked.

SEE ALSO

semaphore(2)

NAME

crossInvoke – cross-context invocation

DECLARATION (In sos.h)

```
class invokeMessage {
  ...
 public:
  long opCode;    /* opaque for system */
};

class returnMessage { ... };

  returnMessage*
crossInvoke (invokeMessage*, segmentDesc*[] =0, index =0);
```

SYNOPSIS

```
#include <sos/sos.h>
class myInvokeMessage: public invokeMessage { ... };
class myReturnMessage: public returnMessage { ... };
myInvokeMessage* im;
segmentDesc *segs[];
int index;
long myOpCode ;
...
// Fills the opCode field to select the remote procedure called
im -> opCode = myOpCode ;
myReturnMessage* rm= (myReturnMessage*) crossInvoke (im, segs, index);
...
```

ARGUMENTS

The *im* argument is a pointer to an invocation message.

The *segs* argument is a vector of pointers to segments descriptors (see *segment(2)*). The vector must be terminated by a 0 pointer.

The *index* argument is an index in the *trapReference* table of the calling object (see *sosObject(2)*, *setTrapRef(2)*).

DESCRIPTION

Force a a process to continue its execution in the context of the principal of the calling object, as indicated by the *trapReference* in the latter's acquaintance descriptor. A bitwise copy of the invocation message is transfered and the descriptors for remote access to segements *(segs)* are passed. Only called in sosObject ! **CrossInvoke** is to be called from within a proxy only.

RETURN VALUE

rm is a pointer to bitwise copy of the return message.

EXCEPTIONS

any exception
if the principal terminates with an exception, then *crossInvoke* resignals the same one.

badTrapReference
the principal is not found or the *trapReference* designated by the *index* argument is not set.

crossInvokeException
the invoked context has crashed

LIMITATIONS

The size of invoke and return messages is limited by *MAXMESSAGE*. If they are larger, the result is undefined.

The Return message *(rm)* is valid only until the next **crossInvoke** call in the same task.

BUGS

The address of the calling object is taken from the contents of the stack. This only works if **crossInvoke** is called directly from the object's code (not from a procedure called by the object); also, the calling code may not be *inline*.

SEE ALSO

proxy(2), stub(2), segment(2), setTrapRef(2)

NAME

dkeyof – calculus of the dynamic class type-checking key

SYNOPSIS

- In dyncompiler.h:

```
typedef long KEYTYPE;
```

- In any application:

```
// class declaration
dynamic class A {
  ...
};

F(){
    A* a1;
    A a2;
    KEYTYPE dkey;

    dkey = dkeyof( A );
    dkey = dkeyof( *a1 );
    dkey = dkeyof( a2 );
}
```

DESCRIPTION

Dkeyof returns a 32-bit key (type *long)* which characterizes the dynamic class for run-time type-checking. The *dkeyof* operator is similar to the C operator *sizeof.* This key is automatically passed as an extra argument to the SOS procedures for run-time interface verification between client and server (or provider) declarations.

Returns 0 if the argument is not a class, not a dynamic class and not an instance of a dynamic class. Note that 0 is not a valid key.

SEE ALSO

dynamic(2), DL_INFO(8)

On the use of the dynamic link editor in SOS V4, in ~sos/v4/doc/dlink

NAME

dynamic – introduction to dynamic classes in UNIX and SOS

SYNOPSIS

- const char *codeName;
  const char *providerName;

  // class declaration
  dynamic [(*codeName*)] **class** *className* { ... };

  // object instantiation
  new dynamic [(*providerName*)] *className;*

- const ref *codeRef;
  const ref *providerRef;

  // class declaration (SOS only)
  dynamic [(*codeRef*)] **class** *className* { ... };

  // object instantiation (SOS only)
  new dynamic [(*providerRef*)] *className;*

DESCRIPTION

The keyword **dynamic** in a class declaration means that the code or/and the instance will be dynamically loaded. The code will be then linked with the importer. The *codeName, providerName, codeRef,* and *providerRef* are optional; they are interpreted differently under Unix and under SOS:

- *codeName* is the file name (interpreted with respect to the environment variable *DLAPTH)* containing the code for the class. In UNIX, an argument of type *ref* is an error and a run-time error occurs if *DLPATH* is not found in the application environment.
- In SOS, *providerName* is the name of a provider to be queried for a proxy. The provider must have previously registered this name with the Naming Service.
  ProviderRef is the reference of the provider. The provider must furnish an object of class code (see *code(2), makecode(7)*). When a dynamic instantiation is performed, the proxy is imported and becomes the new instance.

If *DLPATH* is not set, the environment variable *SOS* is searched. If set, the default path is *${SOS}/export.* If not set, this default path becomes */sos/export.*
If *DLPATH* is not declared, the default *codeName* is */export/className.code.*
The default *providerName* is */services/className.*

The actual data migration and/or loading of code are performed by the procedures *sosImport* or *sosFindCode* which are called at instantiation time:

- before any instruction in the block for an **automatic** variable;
- at the allocation statement for an object allocated by **new;**
- before the execution of **main()** for static variables.

If the dynamic linker can't find a procedure, exceptions *noPrincipal, noCode, tableFull, principalError, dynBadInterface* can be raised (see exception(2)).

EXAMPLE

```
// in the header file
dynamic class A {
  A ();
  int m ();
```

```
};

// in the exported file, compiled with sosCC -c
A::A () {
  printf ("constructor reached\n");
}

A::m () {
  return printf ("method m reached\n");
}

// in the importer file, compiled with sosCC
main () {
  A* a;

  begin
     a = new dynamic A;
  except
     when( noCode )
        printf( "class A: code not found\n" );
        exit(1);
     when( dynBadInterface )
        printf( "class A: wrong interface\n" );
        exit(1);
  end

  a->m ();
}
```

FILES

file.c	source file
file.o	export file compiled with sosCC -c

SEE ALSO

sosCC(1), mergexport(1)
Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley 1986.
B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall 1978.
On the use of the dynamic link editor in SOS V4, in ~sos/v4/doc/dlink

BUGS

The exported code for a dynamic class cannot be split into more than one source file. If this case might occur, run mergexport on these different files.

The code for a dynamic class is imported only once; hence it is useless to use different file names for subsequent instantiations.

Inline methods of dynamic classes are not inlined. It is advised not to use them.

DIAGNOSTICS

Dynamic link errors (and internal bugs) begin with the message "Error dynamic link".

NAME

exception – introduction to exceptions in C++

SYNOPSIS

```
#include <sos/sos.h>
   int
my_procedure_1 ( . . . )
   raises ( error1 )
   raises ( error2 )
{
   . . .
   if ( error1_occured ) raise ( error1 );
   . . .
   if ( error2_occured ) raise ( error2 );
}
   void
my_procedure_2 ( . . . )
   raises ( error2 )
   begin
      . . .
      i = my_procedure_1 ( . . .);
      . . .
   except
      when ( error1 ) i = 0;     /* catch error and fix */
      when ( error2 ) reraise(); /* forward error to my caller */
      when (others)  . . .       /* catch other errors */
   end;
}
main () {
   . . .
   begin
      . . .
      my_procedure_2 (. . .);
      . . .
   except
      when ( error2 ) {        /* catch error and fix */
         printf (''error 2 occurred'');
         . . .
         }
   end;
   . . .
}
```

DESCRIPTION

The exceptions package allows a procedure to terminate:

- either normally with a **return** statement, or by falling off the end,
- or with an exception, with a **raise** or a **reraise** statement.

Raise signals an exception to the immediate caller; **reraise** re-signals a caught exception one step further up the call stack. The argument to **raise** is an arbitrary exception name.

Exceptions may be caught by the caller by enclosing the call in a **begin...when...end** block. Exceptions raised by procedures called in the first part of the block may be caught by the **when** clauses. A **when(** *exception_name*) catches a named exception; **when(others)** catches all exceptions not explicitly caught by another **when clause.** An exception caught in a **when** clause may be re-signaled by **reraise.**

Begin...end blocks may be nested. Handlers may raise exceptions.

Exceptions not caught terminate the program with a -1 termination code.

BUGS

If a procedure, returning a non-**void**, contains a **raise** or a **reraise** statement (like *my_procedure_1* in the example above), the compiler will complain about not returning a value. These error messages may safely be ignored.

Exceptions are implemented using preprocessor macros. They have the same limitations as any macro package, in particular, poor error handling.

When a procedure terminates with an exception, **"auto"** objects (created on the stack), between the **begin** matching the exception handler and the call, are destroyed, but their destructor is not called.

SEE ALSO

Vadim Abrossimov, *"Exception handling in C++" in* **~sos/v1/doc/exceptions**

NAME

segmentDesc::getAddr – get start address of (sub)segment

segmentDesc::getSize – get size of (sub)segment

segmentDesc::getMaxSize – get max. size of any (sub)segment possibly referred

segmentDesc::getCapability – get capability list

DECLARATION (in sos.h)

```
   byte*
segmentDesc::getAddr ();
   long
segmentDesc::getSize ();
   long
segmentDesc::getMaxSize ();
   segmentCapability
segmentDesc::getCapability ();
```

EXCEPTIONS

sgNotLocal

the **segmentDesc::getAddr** operation was attempted for a remote segment.

SEE ALSO

segmentDesc(2)

NAME

giveMyOID – give one OID of current instance to another accointance.

DECLARATION (in sosObject.h)

void
sosObject::giveMyOID (const sosObject*, const int)
raises(notAnObject)raises(tableFull)raises(badIndex);

SYNOPSIS

#include <sos/sos.h>
sosObject* *toObject;*
sosObject* *fromObject;*
int *index;*
fromObject->**giveMyOID** *(toObject, index);*

ARGUMENTS

The *toObject* argument is the address of an acquaintance (see *sosObject(2))*, to which an OID of *fromObject* is to be given. The *index* argument is the index of the desired OID, in the OID table of *fromObject*'s acquaintance descriptor.

DESCRIPTION

Add to the OIDs table in the acquaintance descriptor for *toObject*, the fromObject's OID designated by *index.*

EXCEPTIONS

Raises *notAnObject* if the address *toObject* does not refer to a valid acquaintance, *tableFull* if the table of OIDs in the acquaintance's descriptor of *toObject* is full, *badIndex* if the *index* argument is negative or greater than MAX_OIDS - 1.

SEE ALSO

group(2), sosObject(2), ref(2), acquaintanceService(3)

NAME

givePoxy – user procedure called when a proxy is requested.

DECLARATION (in sosObject.h)

```
/* class of description for a proxy to export  */
struct proxyDesc {
  ref proxy;  /*  reference of the given proxy */
  int copy;   /* flag to say if the proxy's data is to be copied
              (true) or moved (false, the default) to
              the client context */
};
   virtual void
sosObject::giveProxy (importRequest* ir, proxyDesc* result)
        raises(refused);
```

SYNOPSIS

```
#include <sos/sos.h>
class myProxyProvider: public sosObject { /* or some class derived from sosObject */
  ...
     void
  giveProxy (const importRequest* ir, proxyDesc* result) {
       ...          /* fill in the proxyDesc structure */
  };
  ...
};
```

ARGUMENTS

The *ir* argument is a pointer on an *importRequest(2)* class or a structure derived of.

The *result* argument is a pointer on a **proxyDesc** structure which is to be filled in.

DESCRIPTION

The *giveProxy* procedure is a user-written procedure. It should construct or select a proxy object for the *client*. The ref(2) to this proxy object is fills *result*. When this procedure returns successfully, the proxy is automatically migrated to the client context. This procedure should be implemented by each class of objects wishing to export proxies. It is called as part of the proxy migration protocol of the Acquaintance Service (see *acquaintanceService(3)*).

The acquaintance descriptor of the given proxy must contain its code reference and its trap reference, to be set by calling the **setCodeRef** and **setTrapRef** operations. The code object associated to the proxy is an instance of class **code** (see *code(2)*). Conversely, the principal may make one of its own Trap References (or that of some other acquaintance of the same group) point to the proxy during the **giveProxy** operation, by calling **setTrapRef.**

The Acquaintance Service will automatically update the trap reference, when migrating the proxy (see *setCodeRef(2), setTrapRef(2)*).

DEFAULT

The default, **sosObject::giveProxy,** always signals **refused** (see *sosObject(2)*).

RETURN VALUE

The *result*.**proxy** return value is the reference of the proxy to migrate.

The *result*.**copy** return value is a flag which indicates if the data segment of the proxy must be copied (true) or moved (false, the default), into the client context.

EXCEPTIONS

Signal **refused** to indicate unwillingness to create the requested proxy.

LIMITATIONS

We are not able to generate automatically the **giveProxy** operation. A library of standard **giveProxy** operations is predefined, containing currently the operations: **giveSelf, giveCopy.** For other cases, one must write one's own migration code (see *proxy(2)*).

SEE ALSO

code(2), importRequest(2), proxy(2), reference(2), setCodeRef(2), setTrapRef(2), sosObject(2), acquaintanceService(3)

NAME

giveTrapRef – give one trap reference of current instance to another acquaintance.

DECLARATION (In sosObject.h)

int
sosObject::giveTrapRef (const sosObject*, const int, const opaqueBytes& = nullOpaque)
raises(notAnObject)raises(notSameGroup)raises(noTrapRef);

SYNOPSIS

#include <sos/sos.h>
sosObject* *toObject;*
sosObject* *fromObject;*
int *index;*
opaqueBytes *myOpaque;*
. . .
fromObject **->giveTrapRef** (*toObject,index,myOpaque*);

ARGUMENTS

The *toObject* argument is the address of an acquaintance (see *sosObject(2)*), to which a trap reference of *fromObject* is to be given. The *index* argument is the index of the desired trap reference, in the trap references table of *fromObject*'s acquaintance descriptor. The *myOpaque* argument is used to fill the opaque field of the new trap reference (see *reference(2), opaqueBytes(2)*). It is an optional argument. If not furnished it takes the default value of a zero-filled opaqueBytes instance.

DESCRIPTION

Add to the trap references table in the acquaintance descriptor for *toObject*, the fromObject's trap reference designated by *index*. This operation is restricted between objects of the same group.

RETURN VALUE

returns the index in trap reference table of *toObject*

EXCEPTIONS

Raises *notAnObject* if the address *toObject* does not refer to a valid acquaintance, *notSameGroup* if *toObject* and *fromObject* do not belong to a common group. *noTrapRef* if the slot designated by *index* in the table of trap references of *fromObject* argument is not a valid one.

SEE ALSO

crossInvoke(2), group(2), sosObject(2), reference(2), opaqueBytes(2)

NAME

group – communication group, distributed object

DESCRIPTION

A group is defined as the set of acquaintances, across contexts, which carry a same OID:

- either all carry a same group OID,
- or one of the acquaintances gives its concrete OID to be used as a group OID by all the others.

SEE ALSO

OID(2), giveMyOID(2), sosObject(2), acquaintanceService(3), addGroupOID(3)

NAME

importRequest – argument class of constructors for imported instances of dynamic classes and of the **giveProxy** Procedure

DECLARATION **(in acquaintanceService.h)**

```
/* the first argument for a giveProxy crossInvocation must be
a pointer to a class
importRequest
or derived from
importRequest */

class
importRequest: public invokeMessage{
   public:
   importRequest(){;}
   importRequest(const ref& reqObj){;}
        radius
   locateClient();
        const ref&
   getClient();
        const ref&
   getReqObj();
};
```

DESCRIPTION

An importRequest is transferred by the Acquaintance Service from the client context to the principal context during an importation. It is used to transport arguments for the *givePproxy* procedure.

The *locateClient* procedure returns the radius from the client context to the current context: **sameContext, sameWorkstation, sameOffice.**

The *getClient* procedure returns the reference of the client context of importation.

The *getReqObj* procedure returns the reference of the requested object.

SEE ALSO

dynamic(1), giveProxy(2), proxy(2)

NAME

newContext – create a new context

DECLARATION (in sos.h)

void
newContext(OID*, const char*, const char);**

SYNOPSIS

#include <sos/sos.h>
newContext **(hisCID*, pathname, args);**

ARGUMENTS

hisCID is the unique OID of the created context.

The *pathname* argument is the Unix pathname of an SOS executable file evaluated using the Unix PATH environment variable.

args are arguments to the new context (see execve(2) in the Unix manual).

DESCRIPTION

The primitive **newContext** can be used to create a new context for executing an SOS application.

After creation, the two contexts execute independently.

The created context may read its arguments in *argv*.

EXCEPTIONS

Raises **UnixCantExec** to indicate the Unix pathname does not correspond to an executable file; the reason is printed on the standard error file.

NAME

OID – object identifier class

DECLARATION (in OID.h)

```
class OID {
  ...
public:
  ...
  OID ();      /* default constructor */
  OID (const OID&); /* copy constructor */
  OID (importRequest *); /* constructor for importated OIDs */
       ulong
  getSite();      /* returns creation site identifier of an OID */
       void
  allocate (); /* allocates a value to a null OID */
       void
  groupAllocate (unsigned long);
       friend char*
  operator=(char* s, const OID& o);
       int
  operator == (const OID&);
       int
  operator != (const OID&);
       void
  print ();
       void
  fprint(FILE*);
  ...
};
```

SYNOPSIS

```
#include <sos/sos.h>
...
OID oid1, oid2;
unsigned long seed;
...
oid1.allocate ();
oid2.groupAllocate (seed);
```

ARGUMENTS

The *seed* argument is an unsigned long less than 1 000 000. It is used as a seed to generate the group; the same seed is guaranteed to always generate the same OID.

DESCRIPTION

An OID is an Object IDentifier. Its possession gives no right on the object it identifies. Its scope is all of a SOMIW universe.

When it is created, an OID has a null value. The *allocate* procedure assigns it a new, unique value in the SOMIW universe. The *groupAllocate* assigns a new group OID value, see *group(2)*.

To give your OID or your group OID to another object, see *giveMyOID(2)*.

SEE ALSO

giveMyOID(2), group(2)

BUGS

The existence of the seed is a bug: it is impossible to guarantee that two independently created groups will actually have different group OIDs. To avoid group OIDs overlapping, it is advisable to refer to group OIDs' list in file /users/sos/v1/groupOIDs, and to update it each time a new group OID is used in some program.

NAME

opaqueBytes – struct for the opaque field of a reference.

DECLARATION (in sos.h)

```
const OIDSIZE = sizeof(OID);
struct
opaqueBytes{
   char area[OIDSIZE];
   opaqueBytes();
   opaqueBytes(opaqueBytes&);
        int
   operator==(const opaqueBytes&);
        int
   operator!=(const opaqueBytes&);
        opaqueBytes&
   operator=(const OID& );
        friend OID&
   operator=(OID& oid, const opaqueBytes&);

};

extern const opaqueBytes nullOpaque;
```

DESCRIPTION

An *OpaqueBytes* struct is an array of *OIDSIZE* bytes. It is the type of the opaque field of a reference (see *ref(2)*). One may fill the opaque field of a reference with the *getReference* procedure of the Acquaintance Service (see *getReference(3)*). It is reserved for the application use, so it is not interpreted by the system.

The overloaded operator = is use to fill an opaqueBytes instance with an OID, or to get an OID from an opaqueBytes instance.

SEE ALSO

reference(2), getReference(3)

NAME

proxy – how to export, how to use, how to compile a proxy

SYNOPSIS

This is an example composed of header files my_principal.h, my_proxy.h defining the interfaces, an exporter my_principal.c and a client my_client.c:

A.

```
/* header file my_principal.h */
/* path of object code of my_principal registered with the Name service */
const char* my_principal_code = "/export/my_principal.code";

dynamic ( my_principal_code ) class my_principal : public sosObject {
  . . .
  // contructor for local instances
   my_principal::my_principal();

  // contructor for imported instances
   my_principal::my_principal( importRequest* );
  . . .
       void
   my_principal::giveProxy( importRequest* , proxyDesc* ) raises(refused);
  . . .
};
```

B.

```
/* header file my_proxy.h */
/* path of object code of myProxyClass registered with the Name service */
const char* myProxy_code = "/export/myProxy.code";

struct myProxyArgs: public importRequest{
        int i;
        myProxyArgs(int x){x = i;}
}

dynamic ( myProxy_code ) class myProxyClass : public sosObject {
  . . .
  //contructor for local instances
   myProxyClass::myProxyClass( int );

  // contructor for imported instances
   myProxyClass::myProxyClass( myProxyArgs* );
  . . .
};
```

C.

```
/* How to export a proxy, file my_principal.c) */
#include <sos/sos.h>
#include "my_principal.h"
#include "my_proxy.h"
code * code_proxy;
```

```
/*
 * The constructor for the principal (for instance) sets up
 * the appropriate group and registers itself with the Name Service
 */

my_principal::my_principal(){
   OID o;
   ulong ul;

        . . .
   o.groupAllocate (ul);

  // set my group OID
   AS->addGroupOID (this, o);

  /* create a code object for future exportations of proxies,
   the arguments of the code constructor are the class name
   followed by the object files list ended by zero */

   code_proxy = new code ("myProxyClass", "my_proxy.o", 0);

  // register its reference with the name service
   NS->add ("my_principal", this);
}

/* The procedure to create a proxy */

   void
my_principal::giveProxy(importRequest* ir, proxyDesc* result)
        raises( refused ){
   ref proxyCode;
   bool notOK;
   myProxyArgs * args = (myProxyArgs*) ir;

   . . .
  // decide whether to export or not
   if (notOK) raise(refused);

  // get reference of proxy code object
   AS->getReference (code_proxy, &proxyCodeRef);

  // create a local instance of proxy using arguments
   myProxyClass* exportProxy = new myProxyClass ( args->i);

  // give my group OID (index 1) to the proxy
   giveMyOID (exportProxy, 1);

  // set proxy's code object reference
   exportProxy -> setCodeRef (proxyCodeRef);

  // if the principal wants to cross-invoke the proxy:
   this -> setTrapRef (exportProxy);
```

```
  // if the proxy needs to cross-invoke its principal:
    exportProxy -> setTrapRef (this);

  // prepare the proxy to migrate, fills the result argument
    exportProxy -> giveSelf (result);
}
```

D.

```
/* How to use a proxy (file my_client.c) */
#include "my_proxy.h"
main(){
   . . .
   const char* the_principal = "my_principal";
   myProxyClass* myProxy;

   . . .
   myProxyArgs args;
   args.i = 7;
   . . .
  /* call constructor for imported instances
   * raises exception "refused","not found", "noCode","noPrincipal"
   *  or "tableFull" when importation fails
   */
   myProxy = new dynamic (the_principal) myProxyClass(& args);
   . . .
}
```

E.

How to compile the different parts of the program (sample Makefile):

```
all: my_proxy.o my_principal my_client

my_proxy.o: my_proxy.h my_proxy.c
        sosCC -c my_proxy.c

my_principal: my_principal.o my_proxy.o my_main.o
        sosCC -o my_principal my_main.o

my_main.o: my_main.c my_principal.h
        sosCC -c my_main.c

my_principal.o: my_principal.h my_principal.c
        sosCC -c -c my_principal.c

my_client: my_client.c my_proxy.h
        sosCC my_client.c
```

DESCRIPTION

In order to migrate an object, it must be of a class both: (1) derived from sosObject; (2) declared *dynamic*.

The argument of the *dynamic* declaration is the symbolic name (or the ref(2)) of the code object for the

dynamic class. For a local instanciation, if there is no such argument and if the code is not linked statically with the application, the code object will be searched under the default name /**export**/*className*.**code**.

A code object is created either at run-time, as in the my_principal constructor example above, or by saving it with the OSS, by running *makecode* under SOS (see *makecode(7)*).

A proxy is an *imported instance* and the declarer is its client. Instanciating an object with the keywords *new dynamic* triggers an importation. The argument of the *dynamic* instanciation is either the name of a principal, registered with the Name Service, or a reference(2). The principal is asked for a proxy, which is then imported, becoming the new instance. For a *dynamic* instanciation, if there is no such argument, the principal will be searched under the default name /**services**/*className*.

The first argument of an importation constructor must be a pointer to an importRequest structure (or derived) (see *importRequest(2)*).

The actual scenario, when executing the sample program above, is the following:

- Before anything, the principal registers its name with the Name Service, and creates a *code* object to hold the code of class *myProxyClass*. The code is loaded from a Unix file; the name of this file is declared in the *code* constructor, and evaluated relative to the environnement variable *DLPATH*.

At the time of the instanciation of *myProxy:*

- The principal's **giveProxy** procedure is called with the *importRequest* structure of the client's context passed as an argument. This terminates:

  * either with the *refused* exception (which is resignaled to the client; in this case, the following steps are skipped),

  * or by normal termination, declaring a proxy *exportProxy* and its associated code object.
- The data for *exportProxy* and (if necessary) its code object are imported into the client's context.
- *myProxy* is instantiated by calling its appropriate constructor. A constructor for a *dynamic* instantiation must have as its first (and possibly only) argument, an *importRequest* (or derived from) pointer.

EXCEPTIONS

An importation request can fail with one of the following exceptions:

refused
if the principal's **giveProxy** procedure refuses to give a proxy to the client.

noPrincipal
when the reference for principal is null or obsolete and its update has not succeeded.

noCode
When the importation of one of the code objects for the proxy has failed.

tableFull
if the table of acquaintances' descriptors is full and the proxy cannot be registered with it.

noNameServer
when there is no instance of the Name service to search the reference of the principal designated by a symbolic name.

principalError
if the proxy reference returned by *giveProxy* has not been filled.

COMPILATION

The code for dynamic classes must be compiled by the *sosCC* command with the *-c* option (see *sosCC(1)*). The produced code is suitable for both static or dynamic linking.

EXAMPLES

Please refer to the examples in ~sos/v2/examples.

SEE ALSO

CC(1), dynamic(1), sosCC(1), crossInvoke(2), importRequest(2), stub(2), giveProxy(2), code(2), setCodeRef(2), setTrapRef(2), acquaintanceService(3), addGroupOID(3), nameService(5), makecode(7)

BUGS

"Static" instances of dynamic classes are not allowed. If you declare an instance of a dynamic class "static" it cannot be initialized properly and behaviour is undefined.

Error lines in the sosCC compilation can be erroneous and then indicate the current end block rather than the real line number.

DEBUGGING

To debug a dynamic class, it is safer to link its code statically.

CHANGES FROM V1

The syntax and semantics of importation has changed:

- in class declaration:

" *dynamic* (*searchname*) class ZZZ {...};"

the *searchname* sets the default name of the code object for class ZZZ.

- in class instanciation:

"ZZZ z1; ZZZ* z2 = *new* ZZZ(...);",

z1 and z2 are created locally by execution of the appropriate constructors.

"ZZZ *z3 = *new dynamic* (*P*)ZZZ(*ir* ,...);

z3 is imported from principal *P;* a constructor is applied to the imported data (its first *ir* argument is an *importRequest* or derived from) pointer.

NAME

ref - reference class

DECLARATION

```
/* hint of location class */

class HINT{
   public:
        HINT(); /* default constructor */
        HINT( const HINT&); /* copy constructor */
        HINT(importRequest* ); /* constructor for imported HINTs */

        OID context;
        OID site;
        short descNum;
        int
   operator==(const HINT&);
        int
   operator!=(const HINT&);
};

extern const HINT nullHINT;
class ref{
   public:
   ref(); /* default constructor */
   ref(const ref&); /* copy constructor */
   ref(importRequest* ); /* constructor for imported refs */
        void
   getOID(OID*);
        void
   getCID(OID*);
        void
   getSID(OID*);
        short
   getDescNum();
        /* provisional */
        void
   setNHreference(const OID& oid);
        void
   print(FILE* fd = NULL);
        int
   operator==(const ref&);
        int
   operator!=(const ref&);
        void
   getOField(opaqueBytes* field);
};

extern const ref nullRef;
```

DESCRIPTION

A reference is an indication about the location of an acquaintance. It is constitued by an object identifier (see *OID(2)*) , a hint of location and an opaqueBytes struct which is an opaque field that may be used by services (see *opaqueBytes(2)*).

The HINT field can be obsolete.

The *getOID* procedure returns the OID of the reference. This OID can be either a concrete OID, or a group one (see getReference(3)).

The *getCID* procedure returns the context OID in the hint of the reference.

The *getSID* procedure returns the site OID in the hint of the reference.

The *getDescNum* procedure returns the index of the acquaintance descriptor in the hint of the reference.

The *setNHreference* procedure sets the OID of the reference to the value of the argument *oid*, and the HINT of the reference to a null value. It is used to give a unique reference to a distributed service. When it encounters a null hint reference, the Acquaintance Service try to find the nearest object which has an OID oid and fills the hint with its location.

The *print* procedure prints the reference on the standard output when its argument *fd* is the default one.

The *getOField* procedure fills the *field* argument with the opaque field of the hint.

SEE ALSO

OID(2), opaqueBytes(2), getReference(3)

NAME

segmentDesc - descriptor for (sub)segment manipulation

DECLARATION (in sos.h)

```
class segmentDesc {
 ...
public:
   segmentDesc( long maxSize= 0, segmentCapability cap= 0 );
      void
   assign( byte* address, long size );
      void
   assign( segmentDesc* baseDesc, long offset= 0, long size= 0 );
      void
   allocate( long size= 0 );
      void
   free();
      void
   copyTo( segmentDesc* targetSegmentDesc );
      void*
   getAddr();
      long
   getSize();
      long
   getMaxSize();
        segmentCapability
   getCapability();
};
```

DESCRIPTION

A segment descriptor is used to refer (sub)segments.

Segment descriptor operations *segmentDesc::allocate(2)* and *segmentDesc::free(2)* allow to allocate/deallocate segments

Segment descriptor operation *segmentDesc::copyTo(2)* allows to copy a (sub)segment into another one

Segment descriptors may be transferred to another context by *crossInvoke(2)*. The tranferred descriptor is used in the target context to refer to the (sub)segment in the source context (which we call "remote (sub)segment"). The operations allowed in the target context may be limited by using the associated capability bit mask.

CONSTRUCTOR(S)

The *maxSize* argument allows to limit size of the (sub)segment which will be referred by the descriptor. If *maxSize* is 0, the size of referred (sub)segment is not limited.

The *cap* argument is a capability bit mask with the following fields:

sgAllocate
: *segementDesc::allocate(2)* operation allowed

sgFree *segmentDesc::free(2)* operation allowed

sgCopyFrom
: *segmentDesc::copyTo(2)* operation allowed

sgCopyTo
: the segment descriptor can be the target descriptor of a *segmentDesc::copyTo(2)* operation

The following operations can always be invoked and are not protected by a capability:

segmentDesc::getSize(2)

segmentDesc::getMaxSize(2)

segmentDesc::getCapability(2)

The following operations can never be invoked for a remote segment:

segmentDesc::getAddr(2)

segmentDesc::assign(2)

SEE ALSO

allocate(2), assign(2), copyTo(2), crossInvoke(2), getAddr(2), sosObject(2)

NAME

semaphore– semaphore class for objects synchronization

DECLARATION (in semaphore.h)

```
class semaphore{
   ...
 public:
   semaphore() ;
   ~semaphore();
   void P();
   void V();
   int isLocked();
};
```

SYNOPSIS

```
#include <sos/semaphore.h>

semaphore sem;
   ...
   sem.P();
   ...
   if (sem.isLocked)
      sem.V();
   ...
```

DESCRIPTION

The *P* procedure allows to acquire one lock at a time. If the semaphore is already locked, the current task is pending until the lock is released.

The *V* procedure allows to release the lock.

The *isLocked* procedure returns non-zero if the semaphore is locked.

SEE ALSO

countLock(2)

NAME

setCodeRef – update code reference of an acquaintance descriptor

DECLARATION (in sosObject.h)

void
sosObject::setCodeRef (ref&);

SYNOPSIS

#include <sos/sos.h>
ref *itsCodeReference;*
sosObject **myObject;*
. . .
myObject -> **setCodeRef** *(itsCodeReference);*

ARGUMENTS

The *itsCodeReference* argument is a **reference** (see *ref(2))* to a **code** object representing the code for *myObject* (see *code(2)).*

DESCRIPTION

Fills the code reference field in the descriptor of the acquaintance, *myObject,* with the argument *itsCodeReference,* (see *acquaintanceService(3)).*

SEE ALSO

sosObject(2), reference(2), acquaintanceService(3)

NAME

setTrapRef – set Trap Reference of an acquaintance

DECLARATION (in sosObject.h)

int
sosObject::setTrapRef (const sosObject*, const opaqueBytes& = nullOpaque);

SYNOPSIS

#include <sos/sos.h>
sosObject* *toObject;*
sosObject* *myObject;*
opaqueBytes *myOpaque;*
. . .
myObject -> **setTrapRef** (*toObject* , *myOpaque*);

ARGUMENTS

The *toObject* argument is the address of an acquaintance (see *sosObject(2)).*

The *myOpaque* argument is used to fill the opaque field of the new trap reference (see *reference(2), opaqueBytes(2)).* It is an optional argument. If not furnished it takes the default value of a zero-filled opaqueBytes instance.

DESCRIPTION

Fills the **trapReference** field of *myObject* with the reference to *toObject;* i. e. causes future cross-invocations performed by *myObject* to be directed to *toObject* (see *crossInvoke(2)).* It is verified that the acquaintances designated by *toObject* and *myObject* are in the same group (see *group(2)).* If the acquaintance designated by *toObject* migrates to an other context, any Trap Reference pointing to it, in *the current context* will automatically be updated to the new location.

RETURN VALUE

returns the index in trap reference table of *myObject*

EXCEPTIONS

Raises *notAnObject* if *toObject* does not refer to a valid acquaintance.

Raises *notSameGroup* if the acquaintances designated by *myObject* and *toObject* do not belong to a common group.

SEE ALSO

crossInvoke(2), sosObject(2), reference(2), group(2), opaqueBytes(2), acquaintanceService(3)

NAME

sosFindCode – Search the dynamic table of a dynamic class

DECLARATION

(int (**)()) sosFindCode
(KEYTYPE dkey, char* className, ref* codeRef);

SYNOPSIS

#include "dyncompiler.h"

KEYTYPE *dkey*;
ref* *codeRef*;
char* *className*;
int *(**dtbl)()*;

dtbl= **(int(**)()) sosFindCode (dkey, className, codeRef);**

ARGUMENTS

The *dkey* argument is filled by a previous call to *dkeyof*(2). A null key disables run-time interface verification.

The *className* argument is the name of the dynamic class of the requested code.

The *codeRef* argument is assumed to be the code reference.

Default references are used when null reference *codeRef* is received as argument (see *dynamic(2)*.

DESCRIPTION

The *sosFindCode* procedure is inserted by the SOS C++ compiler before any call to a non-dynamic constructor of a dynamic class.
If the code is not already present in the caller context, the corresponding code object is imported (loaded and linked). The returned value is a pointer to the dynamic table for this class.

RETURN VALUE

The *dtbl* return value is a pointer to the dynamic table of class *className*.

EXCEPTIONS

The exception *noCode* is raised if the object file cannot be found.

The exception *dynBadInterface* is raised while if run-time interface verification fails.

SEE ALSO

dynamic(1), code(2), proxy(2), ref(2), sosImport(2), sosLookup(2),
"On the use of the dynamic linker in SOS V4", in the directory doc/dlink of the current distribution.

BUGS

The returned value must be cast to *(int (**)())* type. A C++ (version 1.[0-2]) bug does not allow to declare *sosFindCode* as a procedure returning such a type. So *sosFindCode* is declared as int* type.

NAME

sosImport – Imports data, and pre-requisites code of object.

DECLARATION

```
int* sosImport
    (KEYTYPE dkey, int (*** dtbl)(), void* impReq,
        char* className, ref* codeRef, ref* providerRef);
```

SYNOPSIS

```
#include "dyncompiler.h"

KEYTYPE        dkey;
int            (**dtbl)();
importRequest* impReq;
char*          className;
ref*           codeRef;
ref*           providerRef;

sosObject* data = (sosObject*) sosImport
    (dkey, &dtbl, impReq, className, codeRef, providerRef);
```

ARGUMENTS

The *dkey* argument is filled by a previous call to *dkeyof (2).* A null key disables run-time interface verification.

The *dtbl* argument is the address of the dynamic table of procedures for the imported object, the *proxy,* that the *sosImport* procedure has to fill.

The *impReq* argument is an the address of an importRequest structure (see *importRequest(2))* which is passed to the giveProxy procedure (see *giveProxy(2))* in the provider context.

The *className* is the name of the dynamic class of the requested object.

The *codeRef* argument is a pointer to the reference of the default code for the requested object taken from the declaration of the class *className.* The code reference can be found by a previous call to *sosLookup (see* sosLookup(2)). This argument is optional as the data descriptor may carry this information.

The *providerRef* argument is a pointer to the reference of the provider of the proxy. The provider reference can be found by a previous call to *sosLookup.*

Default references are used when null references *codeRef* or *providerRef* are received as arguments (see *dynamic(2).*

DESCRIPTION

The *sosImport* procedure is inserted by the SOS C++ compiler before any call to a dynamic constructor of a dynamic class. Consider the following declarations:

```
dynamic class A {
    A( importRequest* );
    ...
};

F(){
    importRequest impReq;
    A* a =  new dynamic ("A_Provider") A (&impReq);
    ...
```

```
}
```

The dynamic instantiation is generated into:

```
F(){
    importRequest impReq;
    A* a;
    A* tmpA;
    int (**tmpPF)();
    ref* tmpR;

    tmpA = (A*) sosImport
            (1992, &tmpPF, &impReq, "A", 0,
               sosLookup (&tmpR,"A_Provider",0) );
    a = (*tmpPF[0]) (tmpA, tmpPF, &impReq);
}
```

The *sosImport* procedure imports the data of a remote object, assuming that the provider lookup succeeds.

It also imports the code object (and other pre-requisites) if not already present in the context, then fills the *dtbl* argument with the address of the dynamic procedure table for this imported object.

RETURN VALUE

The return value *data* is the data address of the proxy, which is cast to the appropriate pointer type.

EXCEPTIONS

The exception *noCode* is raised if the code object for the proxy cannot be found.

The exception *noPrincipal* is raised the provider of the proxy cannot be found.

The exception *principalError* is raised if the provider method, *giveProxy*, returns a null reference to the proxy.

The exception *refused* is reraised in the client context if the provider method, *giveProxy*, raises it.

The exception *tableFull* is raised is the table of acquaintance descriptors is full.

The exception *dynBadInterface* is raised if the run-time interface verification fails.

SEE ALSO

dkeyof(2), dynamic(2), code(2), giveProxy(2), importRequest(2), proxy(2), ref(2), sosImport(2), sosLookup(2)
On the use of the dynamic linker in SOS V4 , in the directory doc/dlink of the current distribution.

BUGS

The returned value must be cast to the appropriate type. Return type of *sosImport* might be declared void*.

NAME

sosLookup – Search the reference of symbolic name

DECLARATION (in <sos/context.h>)

ref* sosLookup(char*, ref*);

SYNOPSIS

```
#include <sos/context.h>
ref* result = new ref;
char* symbolicName;
...
result = sosLookup (result, symbolicName);
```

ARGUMENTS

The *symbolicName* argument is a *SOS* symbolic name.

The *result* argument is a pointer to a reference.

DESCRIPTION

The *sosLookup* procedure retrieves the reference registered under *symbolicName* within the Name Service, fills the *result* argument with the retrieved reference and returns it.

It is called either the *sosImport* procedure, or by the *sosFindCode* procedure to translate a symbolic name for a provider in a reference.

RETURN VALUE

The return value is the *result* argument filled by *sosLookup*.

EXCEPTIONS

The exception *noNameserver* is raised if the the proxy of Name Service is not present in the current context.

The exception *noPrincipal* is raised when there is no reference registered under *symbolicName*.

SEE ALSO

dynamic(1), proxy(2), ref(2), sosFindCode(2), sosImport(2), nameService(5)

NAME

sosObject – generic class for SOS acquaintances.

DECLARATION (in sosObject.h)

```
class sosObject {
  . . .
 protected:
        virtual void
   giveProxy (const importRequest*, proxyDesc*)
        raises(refused);
        void
   setCodeRef (ref&);
        int
   setTrapRef (const sosObject*, const opaqueBytes& = nullOpaque)
        raises(notAnObject)raises(notSameGroup);
        int
   giveTrapRef (const sosObject* other, int index,
      const opaqueBytes& = nullOpaque)
        raises(notAnObject)raises(notSameGroup)raises(noTrapRef);
        void
   giveMyOID (const sosObject*, const int)
        raises(notAnObject)raises(tableFull)raises(badIndex);
 public:
        virtual void
   stub(invokeMessage*, returnMessage*, segmentDesc**)
        raises(refused);
   /* constructor */
   sosObject();
        virtual
  ~sosObject();
        void
   giveSelf(proxyDesc*);
        void
   giveCopy(proxyDesc*);
   . . .
};
```

DESCRIPTION

In SOS, objects known to the system must derive from class **sosObject.** This is necessary to be able to export proxies, to communicate by cross-invocation, etc.

When it is instanciated, an object derived from **sosObject** is automatically installed as an acquaintance in the current context. Conversely, the **sosObject** destructor removes the object from the list of acquaintances.

The **giveProxy** procedure is called by the Acquaintance Service when a proxy of an acquaintance is requested. The default, **sosObject::giveProxy,** always raises exception **refused,** meaning that the delegation of proxy is refused. Classes capable of exporting proxies should redefine **giveProxy.**

The **sosObject::setCodeRef** procedure fills the code reference field in the descriptor of the current instance.

The **sosObject::setTrapRef** procedure fills the trap reference field in the descriptor of the current instance, it returns the index in trap reference table of this descriptor (see The *opaqueField* argument is used to set the opaque field of the trap reference (see *reference(2), opaqueBytes(2)).* It returns the index in trap reference table of the acquaintance's descriptor.

The **sosObject::giveTrapRef** procedure fills the trap reference field in the descriptor of the *other* instance with the trap reference of the current instance designated by the *index* argument. The *opaqueField* argument is used to set the opaque field of the trap reference of the *other* instance (see *reference(2), opaqueBytes(2)).* It returns the index in trap reference table of the *other's* descriptor.

The **sosObject::giveMyOID** procedure gives one OID of the current instance to another acquaintance.

The **sosObject::stub** procedure is the default stub for derived classes which don't redefine the name **stub(2).** It always raises exception **refused,** meaning that all cross-invocations are refused. See *stub(2)*

The **sosObject::giveSelf** procedure prepares the current acquaintance to migrate by filling the **proxyDesc** structure pointed to by its argument (see *giveProxy(2)).* It should be called by **giveProxy.**

The **sosObject::copySelf** procedure authorize the migration of a copy of the current acquaintance by filling the **proxyDesc** structure pointed to by its argument (see *giveProxy(2)).* It should be called by the **giveProxy.**

LIMITATIONS

"Auto" instances of **sosObjects** cannot be migrated; it is therefore more suitable to allocate **sosObject's** by **new().**

Destructors for **sosObjects,** and objects derived from, may be called by the system, during migration for example. So it is not suitable that a destructor cross-invoke an other context, since it may raise indesirable side effects.

SEE ALSO

crossInvoke(2), giveProxy(2), giveMyOID(2), giveTrapRef(2), opaqueBytes(2), setCodeRef(2), setTrapRef(2), stub(2), acquaintanceService(3)

NAME

stub – principal's stub for the reception of cross-context invocations.

DECLARATION (in sosObject.h)

```
class invokeMessage {
 ...
public:
   long opCode;     /* opaque for system */
};
   virtual void
sosObject::stub (invokeMessage*, returnMessage*, segmentDesc**)
        raises(refused);
```

SYNOPSIS

```
#include <sos/sos.h>
class myClass: public sosObject { ... };
class myInvokeMessage: public invokeMessage { ... };
class  myReturnMessage: public returnMessage { ... };
 ...
   void
myClass::stub
  (invokeMessage* im, returnMessage* rm, segmentDesc *segs[]) {
   myInvokeMessage *mim= (myInvokeMessage*) im;
   myReturnMessage *mrm= (myReturnMessage*) rm;

   switch ( mim -> opCode ) { /* select action to perform */
      . . .
   }
}
```

ARGUMENTS

The *im* argument is a pointer to a bitwise copy of the *crossInvoke(2)* invocation message.

The *rm* is a pointer to the memory for the return message.

The *segs* argument is a vector of pointers to the descriptors for remote *segments(2)* access passed by *crossInvoke(2)*. The vector is terminated by a 0 pointer.

DESCRIPTION

A principal's *stub* function is invoked by the kernel, when it is the target of a *crossInvoke(2)* system call. The execution of the *stub* occurs (as if) in a new, dedicated task. During the *stub*'s execution only, the invoke message (**im*), the return message (**rm*) are accessible and the descriptors (*segs*) can be used for remote access to *segment; see(2)*. The return messsage is filled by **stub** and a bitwise copy of it is returned as the result of the *crossInvoke(2)* call.

DEFAULT

The default stub, **sosObject::stub,** always signals exception **refused.**

RETURN VALUE

A copy of the return message is returned to the invoker.

EXCEPTIONS

If **stub** terminates with any exception, it will be transmitted back to the caller of **crossInvoke.**

LIMITATION

The size of invoke and return messages is limited by *MAXMESSAGE*. If they are larger, the result is undefined.

SEE ALSO

crossInvoke(2), proxy(2), segment(2), sosObject(2), task(2), acquaintanceDescriptor(2)

NAME

unixChannel – base class for the Unix I/O in SOS

DECLARATION

```
class unixChannel {
...
protected:
   short physChannel;
      void
   readReq ();
      void
   writeReq ();
public:
   unixChannel (int fd);
};
```

DESCRIPTION

The **unixChannel::unixChannel()** constructor takes a UNIX file descriptor *fd* as its argument. The constructor sets the value of the **physChannel** field equal to *fd*, meaning that subsequent I/O will take place using that file descriptor.

The functions **unixChannel::readReq()** and **unixChannel::writeReq()**, return, respectively, if and when the **physChannel** file descriptor is ready for a read or a write operation. The called task is rescheduled, if necessary.

SYNOPSIS

An example class, *readWrite* performing Unix-like read/write operations:

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

class readWrite: public unixChannel {
 public:
   readWrite (int fd) : (fd)    {}
      int
   read (char* buff, int len )  { readReq();
                                  return ::read (physChannel, buff, len);
                                }
      int
   write( char* buff, int len ) { writeReq();
                                  return ::write (physChannel, buff, len);
                                }
};
```

Now we can use it as follow:

```
...
char buff[SIZE];
int fd = open( "/dev/tty", . . . );
readWrite myChannel (fd);
...
int count= myChannel.read( buff, SIZE );
...
```

If the task is suspended waiting for tty input, another task may run.

NAME

acquaintanceService – acquaintance service class.

DECLARATION (in acquaintanceService.h)

```
class acquaintanceService: public sosObject{
 public:
        void
   getReference (const OID&, ref*, const opaqueBytes& = nullOpaque)
        raises(notFound);
        void
   getReference (const sosObject*, ref*, const opaqueBytes& = nullOpaque)
        raises(notAnObject);
        void
   getReference (const short, ref*, const opaqueBytes& = nullOpaque)
        raises(notAnObject);
        void
   find (const ref&, radius, ref*)
        raises(notFound);
        short
   getDescriptor (const sosObject*)
        raises(notAnObject);
        short
   getDescriptor (const OID&)
        raises(notFound);
        sosObject*
   getAddress (const OID&)
        raises(notFound);
        sosObject*
   getAddress (const short)
        raises(notAnObject);
        sosObject*
   isAnAcquaintance (const short dn);
        sosObject*
   isAnAcquaintance (const OID&);
        sosObject*
   isAnAcquaintance (const sosObject*);
        void
   getOID (const short, OID*)
        raises(notAnObject);
        void
   getOID (const sosObject*, OID*)
        raises(notAnObject);
        void
   getAllOIDs (const short desnum, OID[])
        raises(notAnObject);
        void
   getAllOIDs (const sosObject*, OID[])
        raises(notAnObject);
        radius
   locate ( const ref&);
        void
   stub (invokeMessage*, returnMessage*, segmentDesc**);
        int
```

```
    addGroupOID (const sosObject*, OID&)
        raises(notAnObject)raises(tableFull);
};
extern acquaintanceService* AS;
```

DESCRIPTION

The acquaintance service is the manager for objects which need to be known by the system (to be able to export proxies, or to communicate by cross-invocation, etc.).

Objects known by the system are all derived from **sosObject** (see *sosObject(2)*) and are installed as acquaintances of current context by the acquaintance service.

The acquaintance service maintains information on objects, such as their address, their groups, their *trap reference* (see *crossInvoke(2)*), etc. It manages proxy migration and object search (see *find(3)*) in co-operation with the kernel.

The acquaintance service is a distributed object. On each site, a main acquaitance service context is installed at the boot-time of SOS (see *sos(7), ASmain(7)*). Each newly created context inherits of a proxy of the acquaintance service, identified by the *AS* pointer. This proxy is the local Acquaintance Service of the new context, it is dedicated to the acquaintances of its current context. Each time an operation needs the intervention of several contexts (for exemple, search or migration), the *AS* passes the request to its principal.

SEE ALSO

sos(7), ASmain(7), crossInvoke(2), sosObject(2), stub(2), addGroupOID(3), find(3), getAddress(3), getAllOIDs(3), getDescriptor(3), getOID(3), getReference(3), isAnAcquaintance(3), locate(3)

NAME

addGroupOID – add an OID to an acquaintance's descriptor

DECLARATION (in acquaintanceService.h)

```
int
acquaintanceService::addGroupOID (const sosObject*, OID&)
        raises(notAnObject)raises(tableFull);
extern acquaintanceService* AS;
```

SYNOPSIS

```
#include <sos/sos.h>
sosObject* addr;
OID& oid;
int index;

index = AS -> addGroupOID (addr, oid);
```

ARGUMENTS

The *addr* argument is the address of an acquaintance (see *sosObject(2)).*

The *oid* argument is an OID (see *OID(2)).*

DESCRIPTION

Adds the second argument, *oid,* to the array of OIDs in the descriptor of the acquaintance designated by the fisrt argument, *addr* (see *acquaintanceService(3)).*

RETURN VALUE

Returns the index of *oid* in the array of OIDs in the descriptor of the acquaintance designated by the first argument, *addr.*

EXCEPTIONS

Raises **tableFull** when the array of OIDs of the acquaintance descriptor is full, **notAnObject** if the address does not refer to a valid acquaintance.

LIMITATIONS

The number of OIDs for an acquaintance is limited by the size of the OIDs table in the descriptor.

SEE ALSO

sosObject(2), OID(2), acquaintanceService(3)

NAME

find – inter-context reference search.

DECLARATION (in acquaintanceService.h)

```
void
AcquaintanceService::find (const ref&, radius, ref*)
        raises(notFound);
extern AcquaintanceService* AS;
```

SYNOPSIS

```
#include <sos/sos.h>
ref& ref1;
radius rad;
ref* ref2;
AS -> find (ref1, rad, ref2);
```

ARGUMENTS

The *ref1* argument is some obsolete reference (see *ref(2)*), *possibly obsolete.*

The *rad* argument is the radius of the search.

The *ref2* argument is a pointer to a reference structure.

DESCRIPTION

Searches, within the contexts included in the radius, for an acquaintance matching the reference specified by the first argument. Stores the updated reference in the location pointed by the *ref2* argument.

A *radius* is one of sameContext, sameWorkstation, sameOffice.

The search starts in the calling context and continues (until a match is found) in all the contexts situated within *radius* of the calling context.

RETURN VALUE

The return value *ref2* is a pointer to some reference.

EXCEPTIONS

When the search fails raises exception **notFound.**

SEE ALSO

reference(2)

NAME

getAddress – return the address of an acquaintance.

DECLARATION (in acquaintanceService.h)

```
sosObject*
acquaintanceService::getAddress (const OID&)
        raises(notFound);
sosObject*
acquaintanceService::getAddress (short)
        raises(notAnObject);
extern acquaintanceService* AS;
```

SYNOPSIS

```
#include <sos/sos.h>
short descriptorIndex;
OID& oid;
sosObject* addr;
addr = AS -> getAddress ( descriptorIndex );
addr = AS -> getAddress ( oid );
```

ARGUMENTS

The *descriptorIndex* argument is an index in the acquaintance descriptors table (see *acquaintanceService(3)*).

The *oid* argument is an object identifier, either a concrete OID or a group one (see *OID(3)*).

DESCRIPTION

Returns the address of an acquaintance of this context, indicated by the argument.

RETURN VALUE

The return value *addr* is a pointer to the acquaintance identified by the argument (see *sosObject(2)*).

EXCEPTIONS

Raises **notFound** if the indicated OID does not designate an acquaintance of this context, **notAnObject** if the index does not refer to a valid object.

SEE ALSO

sosObject(2), OID(2), acquaintanceService(3), getReference(3)

NAME

getAllOIDs – return the list of OIDs of an acquaintance.

DECLARATION (in acquaintanceService.h)

```
void
acquaintanceService::getAllOIDs (const short, OID[])
        raises(notAnObject);
void
acquaintanceService::GetAllOIDs (const sosObject*, OID[])
        raises(notAnObject);
extern acquaintanceService* AS;
```

SYNOPSIS

```
#include <sos/sos.h>
short descriptorIndex ;
sosObject* addr ;
OID oidList [MAX_OIDS];
AS -> getAllOIDs ( descriptorIndex, oidList );
AS -> getAllOIDs ( addr, oidList );
```

ARGUMENTS

The *addr* argument is a pointer to an acquaintance (see *sosObject(2)*).

The *descriptorIndex* argument is an index in the acquaintance descriptors table (see *acquaintanceService(3)*). The *oidList* argument is an array of OIDs (see *OID(2)*).

DESCRIPTION

Searches, in the current context, the list of OIDs of an acquaintance identified by the first argument. Stores the list in the array pointed by the *oidList* argument. The first OID in the array is the *concrete OID* of the acquaintance, the following are its *group OIDs*.

EXCEPTIONS

If the descriptor is unvalid or if the pointer is not the address of a local acquaintance, *getAllOIDs* raises the exception *notAnObject*.

SEE ALSO

sosObject(2), OID(2), acquaintanceService(3), getOID(3)

NAME

getDescriptor – return descriptor index of an acquaintance.

DECLARATION (in acquaintanceService.h)

```
short
AcquaintanceService::getDescriptor (const OID&)
        raises(notFound);
short
AcquaintanceService::getDescriptor (const sosObject*)
        raises(notAnObject);
extern acquaintanceService* AS;
```

SYNOPSIS

```
#include <sos/sos.h>
OID& oid ;

sosObject* addr ;
short descriptorIndex;

 descriptorIndex = AS -> getDescriptor ( oid );
 descriptorIndex = AS -> getDescriptor ( addr );
```

ARGUMENTS

The *oid* argument is an object identifier, either a concrete OID or a group OID (see *OID(3)*).

The *addr* argument is a pointer to an acquaintance (see *sosObject(2)*).

DESCRIPTION

Return the descriptor index of the acquaintance in the current context, identified by the argument.

RETURN VALUE

The return value *descriptorIndex* is an index in the acquaintance descriptors table of the current context (see *acquaintanceService(3)*).

EXCEPTIONS

Raises **notFound** if the indicated OID does not designate an acquaintance of this context, **notAnObject** if the address does not refer to a valid object.

SEE ALSO

sosObject(2), OID(2), acquaintanceService(3)

NAME

getOID – return OID of an acquaintance.

DECLARATION (in acquaintanceService.h)

```
    void
acquaintanceService::getOID (short, OID*)
        raises(notAnObject);
    void
acquaintanceService::getOID (const sosObject*, OID*)
        raises(notAnObject);
extern acquaintanceService* AS;
```

SYNOPSIS

```
#include <sos/sos.h>
short descriptorIndex ;
sosObject* addr ;
OID* oid ;
AS -> ( descriptorIndex, oid );
AS -> ( addr, oid );
```

ARGUMENTS

The *addr* argument is a pointer to an acquaintance (see *sosObject(2)*).

The *descriptorIndex* argument is an index in the acquaintance descriptors table (see *acquaintanceService(3)*).

The *oid* argument is a pointer to some OID (see *OID(2)*).

DESCRIPTION

Searches, in the current context, the *concrete OID* of an acquaintance identified by the first argument, and stores it in the location pointed by the *oid* argument.

EXCEPTIONS

If the descriptor is invalid or if the pointer is not the address of a local acquaintance, **getOID** raises **notAnObject.**

SEE ALSO

sosObject(2), OID(2), acquaintanceService(3), getAllOIDs(3)

NAME

getReference – return reference of an acquaintance.

DECLARATION (in acquaintanceService.h)

```
void
acquaintanceService::getReference (const OID&, ref*,
                                   const OpaqueBytes& = nullOpaque)
        raises(notFound);
void
acquaintanceService::getReference (short, ref*
                                   const OpaqueBytes& = nullOpaque)
        raises(notAnObject);
void
acquaintanceService::getReference (const sosObject*, ref*
                                   const OpaqueBytes& = nullOpaque)
        raises(notAnObject);
extern acquaintanceService* AS;
```

SYNOPSIS

```
#include <sos/sos.h>
short descriptorIndex ;
OID& oid ;
sosObject* addr ;
ref* refp ;
AS -> getReference ( descriptorIndex, refp );
AS -> getReference ( oid, refp, field );
AS -> getReference ( addr, refp );
```

ARGUMENTS

The *descriptorIndex* argument is an index in the acquaintance descriptors table (see *acquaintanceService(3)).*

The *oid* argument is an object identifier. It can be either the concrete OID or a group OID of an acquaintance.

The *addr* argument is a pointer to an acquaintance (see *sosObject(2)).*

The *refp* argument is a pointer to a reference structure.

The *field* argument is a C++ reference to an opaqueBytes structure (see *opaqueBytes(2)).*

DESCRIPTION

Creates the acquaintance reference of the object designated by the first argument and stores it in the location pointed by *refp.* If the first argument is an OID, it will be the *identifier* field of the created reference (see *ref(2)).*

If an opaqueBytes structure is passed as an argument, fills with it the opaqueField field of the reference pointed to by *refp.*

EXCEPTIONS

Raises **notFound** if the indicated OID does not designate an acquaintance of this context, **notAnObject** if either the index or the address does not refer to a valid object.

SEE ALSO

sosObject(2), opaqueBytes(2), reference(2), acquaintanceService(3)

NAME

isAnAcquaintance – check if an object is an acquaintance.

DECLARATION (in acquaintanceService.h)

```
sosObject*
isAnAcquaintance (const short);
sosObject*
isAnAcquaintance (const OID&);
sosObject*
isAnAcquaintance(const sosObject*);
extern acquaintanceService* AS;
```

SYNOPSIS

```
#include <sos/sos.h>
short descriptorIndex ;
OID& oid ;
sosObject* addr1 ;
sosObject* addr2 ;
addr2 = AS -> isAnAcquaintance ( descriptorIndex );
addr2 = AS -> isAnAcquaintance ( oid );
addr2 = AS -> isAnAcquaintance ( addr1 );
```

ARGUMENTS

The *descriptorIndex* argument is an index in the acquaintance descriptor table (see *acquaintanceService(3)).*

The *oid* argument is an object identifier, either a concrete OID or a group one see *OID(2)).*

The *addr1* argument is a pointer to an acquaintance (see *sosObject(2)).*

DESCRIPTION

Checks if the object designated by the argument is an acquaintance.

RETURN VALUE

Returns the address of the checked acquaintance if true, NULL if false.

EXCEPTIONS

None

SEE ALSO

sosObject(2), OID(2), acquaintanceService(3), getAddress(3)

NAME

locate – evaluate acquaintance location.

DECLARATION (in acquaintanceService.h)

radius
acquaintanceService::locate (const ref&);
extern acquaintanceService* AS;

SYNOPSIS

#include <sos/sos.h>
ref& *ref;*
radius *rad;*
rad = **AS -> locate** (*ref*);

ARGUMENTS

The *ref* argument is a pointer to an object reference to be located.

DESCRIPTION

Returns the approximate distance between the caller and the location hinted at in the *ref* argument: one of **sameContext, sameWorkstation, sameOffice.**

RETURN VALUE

The *rad* argument is an indication of location.

EXCEPTIONS

None.

LIMITATIONS

The *ref* argument is not checked for validity.

NAME

storageService - management of permanent composite SOS objects.

DESCRIPTION

The storage service permits user to define permanent, possibly composite objects.

Permanent means that they can be stored on disk and later retrieved.
In order to have a permanent representation, an object's class must derive from class **permObject** , which derives itself from **sosObject** (see *permObject(4)).*

A *composite* SOS object is made of multiple segments: its direct segment and one or more segments called indirect segments. All these segments must however be linked together by the mean of a special generic type of pointer called a **permPtr** structure (see *permPtr(4)).*

Creating a permanent representation is made by calling the **permObject::checkpoint()** method. Activation of a permanent object is managed by the Acquaintance Service and the Storage Service, in a way which generalizes migration, a disk being thought of as a very big memory context.

When an object is activated, only its direct segment is mapped. The other segments are mapped on demand, i.e when they are accessed. Detection of "segment fault" is part of the role of the storageObject that is automatically associated with your object (if deriving from class permObject).

SEE ALSO

acquaintanceService(3) permObject(4) permPtr(4) storageObject(4)

NAME

permObject - permanent SOS object class.

DECLARATION (in storageService.h)

```
dynamic class permObject: public sosObject {
        permObjPtr directSeg;
      ulong     ssSite;
        storageObject * SS;
 public:
  permObject();
  permObject(importRequest * ir);
     virtual
 ~permObject();
     void
  initC();
     void
  initA();
     virtual void
  checkpoint();
       permPtr *
  getHead();
     virtual void
  updateReference(ref* oldRef);
};
```

DESCRIPTION

Permanent SOS objects must derive from class permObject. In this way, the object is automatically associated with a **storage object** (see *storageObject(4)*). It contains mainly the head of the list of all permPtrs that make up the object segment structure.

After the constructor has been called, make an explicit call to procedure permObject::initC() for a brand new object, or to procedure permObject::initA() in the case of an activation.

Procedure permObject::getHead() returns the head of the list of all permPtrs of your object (see *permPtr(4)*).
It is useful when setting a permPtr to a 'first level' indirect segment (see permPtr::setPtr(...)).

SEE ALSO

storageService(4) permPtr(4) storageObject(4)

NAME

permPtr - permanent pointer to indirect segment.

DECLARATION (in <sos/permPtr.h>)

```
class permPtr {
        void * ptr;
unsigned int   size;
        OID   accessor;
        OID   segoid;
  protected:
virtual void*
   access(permPtr& target);

  public:
   permPtr();
   permPtr(importRequest * ir);
  ~permPtr();

      void
   setPtr(const permPtr& prev,void * myseg,unsigned int size,OID access);
      void
   setPtr(const permPtr& ,const permPtr& );
      void
   insertPtr(permPtr& prev,void* myseg,unsigned int size,permPtr& p);
      void
   mark();
        int
   operator==(const permPtr& p);
        int
   operator!=(const permPtr& p);
   operator void*();

};
```

DESCRIPTION

A *permPtr* structure defines a permanent pointer to an untyped segment.It keeps not only the effective pointer to the user data segment mapped in a given context, but also a context-independent identification. Links between all **permPtrs** are managed in order to make the overall structure of the object comprehensive for its associated storage manager (see *storageObject(4)).*

The **permPtr::setPtr** procedure sets the permPtr to point to the given segment.The first argument refers to the permPtr that point to the segment in which this permPtr is defined, i.e its father in the tree segment structure (or permPtr::getHead() if it is in the direct segment). A segment is not part of the object structure until this method have been called.

The effective pointer of such a structure being not typed, the generic type **gpermPtr** as been programmed as a macro (in <sos/gener.h>). The following instructions show how to define a new type deriving from permPtr:

```
#include <sos/gener.h>
#include "myclass.h"

typedef myclass * myclassp;
declare(gpermPtr,myclassp);
typedef gpermPtr(myclassp) myclassPtr;
```

The **permPtr::operator void*()** defines a cast operation that permit users to access the effective pointer. This operation is redefined for derived classes. In our above example, the myclassPtr::operator myclass*() has been declared, so that access to the effective pointer contained in a myclassPtr structure can be done in the following manner:

```
myclassPtr ptr;
myclass * mc;

mc=ptr; // equivalent to mc=(ptr::operator myclass*());
```

At the time this method is called a check is made to be sure the segment it is pointed to is mapped. If not the segment is loaded in the context by the associated **storageObject.**

Making changes in the myclass structure cannot be detected by the system. It is the user responsability to tell the system that a given segment have been modified (if you want the changes to take place permanently). This is done by calling the (associated) permPtr::mark() procedure (see *permObject(4)).*

SEE ALSO

storageService(4) storageObject(4) permObject(4)

NAME

storageObject - storage service proxy.

DECLARATION (in <sos/oss/OSp.h>)

```
class getObject : public importRequest{
  public:
    OID which;
    OID clientCtxt;

    getObject(OID wo,OID co) {which=wo;clientCtxt=co;}
};
dynamic class storageObject : public sosObject {
 public:
   storageObject(getObject * ir);
   storageObject(OID whichObject);
  ~storageObject();
      void
   init();
      void *
   map(permPtr& target);
      void
   chekpoint();
      void
   close();
};
```

DESCRIPTION

A *storageObject* is a proxy delivered by the storage service. One such object is automatically imported when creating (or activating) an object defined as deriving from class **permObject** (see *permObject(4)*). A storageObject is concern with the structure of a permanent object in terms of segments (direct or indirects).
Calling its checkpoint() method causes the object's segments that have been modified since the last checkpoint to be written on disk.

When an object is activated, only its direct segment is loaded. The associated storageObject::init() procedure is in charge of marking all the indirect segments as not yet loaded.

Attempts to access an indirect segment which is not loaded is detected by the storageObject. Its map() procedure is then invoked that causes the missing segment to be loaded from disk.

SEE ALSO

storageService(4) permObject(4) permPtr(4)

NAME

nameService – management of symbolic names.

DECLARATION (in dirproxy.h)

```
dynamic class dirs :public sosObject
{
public:

   void
      changeDir(const char*);
   void
      lookup(const char*, ref*);
   void
      addDir(const char*);
   void
      addName(const char*, const sosObject&, const ushort = 0);
   void
      addName(const char*, const ref&, const ushort = 0);
   void
      delName(const char*);
   int
      getDirEntries(const char*, const char*, int);
   void
      print();
   void
      close();
};
```

DESCRIPTION

The name service permits users to refer to objects by symbolic names.It maintains the mapping between these high-level names and object references.

Management of names is based on their syntactic structure, i.e component names separated by '/' separators. An initial '/' indicates an absolute name (from the root of the hierarchy); otherwise the name is relative (e.g to the current working directory).

The name service is constituted of a set of name servers; each of these servers handles a part of the name tree. All the proxies of the name service of a same site share a cache which asociates names to the servers which manage the corresponding objects or sub-trees.

SEE ALSO

addName(5), addDir(5), changeDir(5), lookup(5), getDirEntries(5).

NAME

addAlias – add an alias object in the *SOS* naming tree.

DECLARATION (in dirproxy.h)

void
addAlias(const char *, const char *);

SYNOPSIS

#include <sos/context.h>
char * *aliasName*;
char * *name*;
NS -> addAlias (*aliasName*, *name*);

ARGUMENTS

The *aliasName* argument is the name of the alias to be added.

The *name* argument is the name of the referenced object.

DESCRIPTION

If the *name* argument already exists, an alias entry is created that acts as an alternative name for its referent. In particular, during name resolution, when such an entry is encountered, the process is continued at the referent entry. An alias is an hard link, it means that the alias object contains the reference of the referenced object.

RETURN VALUE

No return value

EXCEPTIONS

Raises **nameExists** if the directory entry already exists.

Raises **nsBadDir** if a pathname (for *aliasName* or *name*) doesn't refer to a valid path in the naming tree.

Raises **notFound** is the *name* argument doesn't already exist.

SEE ALSO

nameService(5), addName(5), lookup(5).

NAME

addDir – add a directory in the *SOS* naming tree.

DECLARATION (in dirproxy.h)

```
void
addDir(const char *);
```

SYNOPSIS

```
#include "context.h"
const char * dirname;
NS -> addDir (dirname);
```

ARGUMENTS

The *dirname* argument is the name of the directory to be added.

DESCRIPTION

If server handles a directory with a such name, the directory is created on one of the name servers.

The server which maintains a mapping for the authority of the directory to be created.

RETURN VALUE

No return value

EXCEPTIONS

Raises **nameExists** if the directory entry already exists.

Raises **nsBadDir** if the pathname does not refer to a valid path in the naming tree.

SEE ALSO

nameService(5), addName(5)

NAME

addName – add a symbolic name in the *SOS* naming tree.

DECLARATION (in dirproxy.h)

```
void
addName (const char *, const ref &, const ushort = 0);
void
addName (const char *, const sosObject &, const ushort = 0);
```

SYNOPSIS

```
#include "context.h"
char * symbName;
ref    myref;
sosObject * addr;
int flag;
NS -> addName (symbName, myref);
NS -> addName (symbName, *addr);
NS -> addName (symbName, *addr, flag);
```

ARGUMENTS

The *symbName* argument is the pathname to be added (see *nameService(5)).*

The *myref* argument is a reference of an existing object (see *reference(3)).*

The *addr* argument is a pointer to an acquaintance (see *sosObject(3), acquaintanceService(3)).*

The *flag* argument may indicate the mapping can be overwritten if the name is already registered.

DESCRIPTION

Add, if it doesn't already exist, a directory entry containing the mapping between the symbolic name and the object's reference which can be designated by its reference (first form) or by its address (second form). If the third argument is equal to NS_OVERWRITE, the mapping can be overwritten if the name is already registered.

RETURN VALUE

No return value.

EXCEPTIONS

Raises **nameExists** if the directory entry already exists and the flag is different of NS_OVERWRITE.

Raises **nsBadDir** if the pathname does not refer to a valid path in the naming tree.

Using the second form, the **notAnObject** exception can be generated if the address argument doesn't designate a valid acquaintance.

SEE ALSO

nameService(5), changeDir(5), lookup(5), getReference(3)

NAME

changeDir – change working directory.

DECLARATION (in nameService.h)

```
void
changeDir(const char *);
```

SYNOPSIS

```
#include "sos.h"
char * dirname;
NS -> changeDir (dirname);
```

ARGUMENTS

The *dirname* argument is the pathname of a directory object (see *nameService(5)).*

DESCRIPTION

Upon successful completetion, this will cause the named directory to become the current working directory, i.e the starting point for relative pathname searches. After a name service proxy has been imported,the working directory is set to the root

RETURN VALUE

No return value

EXCEPTIONS

If the name doesn't exist, the **notFound** exception is raised.

If the pathname doesn't begin with a '/' and the working directory has not been previously set, the **NSnoCurDir** exception is raised.

SEE ALSO

nameService(5), lookup(5).

NAME

openDir – open a directory in the *SOS* naming tree.

readDir – return the description of a directory entry.

closeDir – close a directory previously opened by openDir.

DECLARATION (in dirproxy.h)

```
class description {
 public:
   description();
   char * getName();
   char * getTypeName()
   int isAlias();
   int isNode();
   int isLeaf();
   void print();
   void printType();
}


     void
openDir (const char * ,int *);

     void
readDir(const int&, description * );

     void
closeDir (const int& );
```

SYNOPSIS

```
char * symbName;
int   idDir;
NS -> openDir (symbName, &idDir);


description desc;
NS -> readDir (id ,&desc);

NS -> closeDir(id);
```

ARGUMENTS

The *symbName* argument is the symbolic name of the directory to be opened.

The *idDir* argument is an int descriptor for the directory.

The *desc* argument is a description of an entry.

DESCRIPTION

openDir opens the directory named by symbName and fills an int descriptor to be used to identify the directory in operations such as readDir and closeDir.

readDir reads the next directory entry of a directory previously opened by **openDir**; it fills a description structure which contains some attributes of the object corresponding to the entry.

closeDir closes a directory previously opened by **openDir**.

RETURN VALUE

No return value

EXCEPTIONS

Raises **nsBadDir** if the pathname doesn't refer to a valid path in the naming tree (openDir).

Raises **nameNotExists** if the directory entry doesn't exist (openDir).

Raises **openOverflow** if you try to open more than five directories (openDir).

Raises **notADirectory** if the name exists but doesn't refer to the name of a directory (openDir)

Raises **notOpen** if the int descriptor doesn't refer to an opended directory (readDir, closeDir).

Raises **noMore** if there isn't no more directory entry (readDir).

SEE ALSO

nameService(5)

NAME

delName – delete a name in the *SOS* naming tree.

DECLARATION (in dirproxy.h)

```
int
delName (const char *);
```

SYNOPSIS

```
#include "context.h"
char * symbName;
NS -> delName (symbName);
```

ARGUMENTS

The *symbName* argument is the name to be deleted.

DESCRIPTION

delName deletes the *symbName* in the naming tree.

RETURN VALUE

No return value.

EXCEPTIONS

Raises **notFound** if the name doesn't exist.

Raises **NSrefused** if the symbName is "/"

SEE ALSO

nameService(5)

NAME

find – find names.

DECLARATION (in nameService.h)

void
find (char * exp, const char * dirname ..., 0);

SYNOPSIS

#include <sos/sos.h>
char * *exp*;
char * *dirname1*;
char * *dirname2*;
NS -> find (*exp*, *dirname1*, *dirname2*, *0*);

ARGUMENTS

The *exp* argument is a description of an object set. The *dirname* arguments are pathnames of the starting point of the search process.

DESCRIPTION

find recursively descends the directory hierarchy for each pathname in the list; if the currently examined object is found to be member of the objectSet described by the 'exp' argument, its name is printed. An objectSet permits to specify assertions on attributes values and to rely these assertions by booleans operators such as +(or), !(not), &(and), -(and not). By now, the only attribute you can use is the name of the object.

RETURN VALUE

No return value

EXCEPTIONS

The **nsBadDir** exception is raised,if a pathname doesn't refer to a valid directory object.

SEE ALSO

nameService(5)

NAME

getDirEntries – look up contents of a directory in the *SOS* naming tree.

DECLARATION (in dirproxy.h)

```
int
getDirEntries (const char *, const char *, const int);
```

SYNOPSIS

```
#include "context.h"
char * symbName;
char * buf;
int n;
NS -> getDirEntries (symbName, buf, );
```

ARGUMENTS

The *symbName* argument is the name of the directory to be looked up.

The *buf* argument is a pointer to the buffer to be filled in.

The *n* argument is the expected number of entries.

DESCRIPTION

getDirEntries fills in the buffer pointed to by *buf* with a maximum number *n* of *NSentry* structures. That structure contains the following fields:

```
ushort    dep;
ushort    length;
ushort    type;
char      name[MAXNAMELENGTH];
```

The *dep* field is an offset to the next structure of the buffer; if it's the last, *dep* contains 0. The *length* field is the length of the entry name. The *type* field is the type of the entry (*NSleaf* or *NSnode*). The *name* field is the entry name being described.

RETURN VALUE

The total number of entries of the directory.

EXCEPTIONS

Raises **notFound** if the directory pathname doesn't exist.

Raises **emptyDir** if the directory is empty.

SEE ALSO

nameService(5), lookup(5)

NAME

lookup – return the reference associated with a name in the *SOS* naming tree.

DECLARATION (in dirproxy.h)

```
void
lookup(const char *, ref *);
```

SYNOPSIS

```
#include "context.h"
const char * name;
ref objref;
NS -> lookup(name, &objref);
```

ARGUMENTS

The *name* argument is the pathname of some object to look up (see *nameService(5)).*

If relative, the name is interpreted relatively to the working directory.

The *objref* argument is a pointer to a reference structure.

DESCRIPTION

The proxy determines the server to invoke by looking up the cache; if it can't find a prefix for the name in the cache the request is broadcasted to all the name servers. If one server maintains the mapping between the name and a reference, the reference structure pointed by *objref* is filled.

EXCEPTIONS

The search can fail, if some component of the pathname does not designate a directory object, or if the final component doesn't exist, raising respectively exceptions **nsBadDir** or **notFound.**

SEE ALSO

nameService(5), addName(5)

NAME

verify – verify whether a given string corresponds to a valid name.

DECLARATION (in nameService.h)

typedef enum {singleMatch, multipleMatch, noMatch} matchType;

matchType
verify(char * descName);

SYNOPSIS

#include <sos/sos.h>
char * *myDescName*;
matchType *res*;
res = NS -> verify (*myDescName*);

ARGUMENTS

The *myDescName* argument is a descriptive name (see *nameService(5)*). It permits users to refer to objects even if they don't know the exact order of the name component making up its symbolic name.

DESCRIPTION

The nameService examines first the pathname part (if not empty) of the *descName* argument; then, each path of the naming tree, beginning from this starting point, is searched for a match to occur with the complete list of unordered name component making up the descriptive part.

When such a match is found, the matching component is suppressed from the initial component list and a report is made. For each path, the search stops if there is no more component in the list or no match can be found.

The name components which composed the descriptive part of the name are separated by ',' separators.

RETURN VALUE

The return value *res* is one of single, multiple or noMatch report.

EXCEPTIONS

The **nsBadDir** exception is raised if the pathname part of the desName argument doesn't refer to a valid directory object.

SEE ALSO

nameService(5)

NAME

createFamily – create a distributed family object

DECLARATION (in rpcMgrProxy.h)

void
rpcMgrProxy :: createFamily (const OID&, const char*, const int = 1)
Raises (tooBig)
Raises (cantAllocate);

SYNOPSIS

#include <sos/context.h>
#include <sos/rpc.h>
dynamic *(/rpcMgr)rpcMgrProxy** **rpc;**
rpc = *new rpcMgrProxy();*
OID *familyOid*
char* *familyName*
int *maxMembers*
rpc->createFamily *(familyOid, familyName, maxMembers);*

ARGUMENTS

The *familyOid* argument is the group OID of the created family object (obtained e.g from groupAllocate(3)).

The *familyName* argument is a character string identifying the created family. This family object registers itself as the server of name *familyName*.

The *maxMembers* argument is the maximum number of this family's members which may exist per host. This argument must be less than *MAXMAXMEMBERS*.

DESCRIPTION

A family is a set of related sosObjects, communicating using some protocol.

It is composed of a set of members and a distributed protocol object called its *familyObject*, which implements communication between family's members.

There is one instance of its *familyObject* per host. Each *familyObject* registers the arrival and/or departure of local members.

The *familyObject* delivers a proxy called familyStub to newly created members.

EXCEPTIONS

Raises *tooBig* if one tries to create a family with maxMembers greater than *MAXMAXMEMBERS*, the maximum size of a family per host.

Raises *cantAllocate* if insufficient ressource are available to create the family.

LIMITATIONS

The *familyOid* argument is not checked for uniqueness. One has to take care that this group OID isn't used elsewhere.

familyName argument is not checked for uniqueness. One has to take care that no other server has the same name.

We limit to *MAXMAXMEMBERS* the number of elements of the same family within a host.

We also limit the number of families which may be created.

SEE ALSO

rpc(6), sosFamilyMember(6), groupAllocate(3)

NAME

fcrossInvoke – family crossInvoke

DECLARATION (in sosFamily.h)

```
returnMessage*
sosFamily::fcrossInvoke(const invokeMessage*,
                        const segmentDesc** = 0,
                        const OID& = nullOID );
Raises (unknownMember);
Raises (noMembers);
Reraises (any remote exception);
```

SYNOPSIS

```
#include <sos/context.h>
#include <sos/rpc.h>
invokeMessage* myInvMess;
segmentDesc** segs;
OID calleeOID;
returnMessage* resmess;
resmess = fcrossInvoke( myInvMess, segs, calleeOID );
```

ARGUMENTS

myInvMess argument is the invocation message. This argument has the same meaning as for the *kernel crossInvoke* call.

segs is the pointer to a null terminated list of segemnt pointers. This argument has the same meaning as for the *kernel crossInvoke* call. Its default value is *NULL*

calleeOID specified which member(s) is (or are) to be called. If nullOID, all members except the caller are to be called according to the protocol of this family. Otherwise, the specified member is called according to the protocol of this family. Its default value is *nullOID*.

DESCRIPTION

This call is used to invoke either one specific member of the family, or the set of all members of the family except the caller.

This call replaces the *kernel crossInvoke(2)* for the family.

The local *familyObject* cooperates with remote *familyObjects* to handle this multicast.

In the case of a multicast, each member executes the call and returns its result to the caller.

Then the *familyObject* collects all replies.

RETURN VALUE

The local *familyObject* returns only the first arrival reply as the result of multicast.

Futher replies are saved and are available to the caller until the next multicast request.

The caller gets each following reply by calling *giveNextReply()*.

EXCEPTIONS

Raises *unknownMember* if the specified calleeOID is unknown within the family.

Raises *noMembers* When one makes a multicast to all family's members and there is no other member than the caller.

Reraises any remote exception if the first reply is an exception.

LIMITATIONS

The *member OID* must identify a unique member of the family. When we add a member, we check that this member is not already declared within this host.

Since the check is not global, one has to take care that different members have different member OID. Otherwise, an invocation with this member OID as calleeOID will cause invocation of an arbitrary member with this memberOID.

This means, we don't support the notion of subset of the family.

SEE ALSO

crossInvoke(2),sosFamilyMember(6),giveNextReply(6)

NAME

giveNextReply – get the next reply of the previous multicast.

DECLARATION (in sosFamily.h)

returnMessage*
sosFamily :: giveNextReply()
Raises (noMoreReply)
Reraises (any remote exception);

SYNOPSIS

```
#include <sos/context.h>
#include <sos/rpc.h>
returnMessage* resmess;
invokeMessage* myInv;
segmentDesc** segs;
resmess = fcrossInvoke( myInv, segs )
while (...) do {
resmess = giveNextReply ();
}
```

DESCRIPTION

This call returns the next reply of a previous multicast.

The call returns immediately if there is any pending reply.

Otherwise the caller is blocked until the availability of a reply.

RETURN VALUE

resmess points to the current returnMessage.

EXCEPTIONS

Raises *noMoreReply* if there is no more reply to return and if the multicast is completed.

Reraises any remote exception if the reply was an exception.

SEE ALSO

sosFamilyMember(6),fcrossInvoke(6)

NAME

giveProtoObj – give a protocol object

DECLARATION (in rpcMgrProxy.h)

void
rpcMgrProxy :: giveProtoObj (const sosObject*, ref *)
Raises (connectionRefused);

SYNOPSIS

#include <sos/context.h>
#include <sos/rpc.h>
dynamic *(/rpcMgr)rpcMgrProxy** **rpc;**
rpc = *new rpcMgrProxy();*
sosObject* *objaddr;*
ref* *trapRef;*
rpc->giveProtoObj *(objaddr, trapRef);*

ARGUMENTS

The *objaddr* argument refers to the caller sosObject.

trapRef points to *trapReference* of the caller i.e the reference of the caller's principal.

DESCRIPTION

This call is available only to the acquaintance service. It request the RPC service to establish a connection between the caller and the callee, by installing one *RPC protocol object* within the caller's station and the other *RPC protocol object* within the callee's station.

RETURN VALUE

Upon return time, the area refered to by *trapRef* is filled with the reference of the RPC protocol object installed within the caller's station.

EXCEPTIONS

Raises *connectionRefused* if for any reason (for instance group mismatch, or unavailable ressource) the connection can't be established.

LIMITATIONS

The number of connections between one station and another is limited.

SEE ALSO

rpc(6)

NAME

rpc – proxy for the RPC service.

DECLARATION (in rpcMgrProxy.h)

```
dynamic class rpcMgrProxy : public sosObject{
 friend class AD;
      void
   giveProtoObj(const sosObject*, ref*);
 public:
      void
   createFamily(const OID&, const char*, const int = 1);
   rpcMgrProxy();
};
```

SYNOPSIS

```
#include <sos/context.h>
#include <sos/rpc.h>
dynamic ( /rpcMgr )rpcMgrProxy* rpc;
rpc = new rpcMgrProxy();
```

DESCRIPTION

The rpc manager is the proxy of the RPC (Remote Procedure Call) service.

Simple point-to-point RPC connections are established implicitly: A normal user doesn't need RPC manager proxy for this.

The RPC manager proxy must be imported to request special functions from the RPC service. Currently, only one such function is defined: *creatFamily* creates a family of objects which communicate using a reliable multicast RPC.

SEE ALSO

createFamily(6), giveProtoObj(6), sosFamilyMember(6)

NAME

sosFamilyMember – the base class for member of a sos family.

DECLARATION (in sosFamily.h)

```
class sosFamily : public sosObject {
 protected:
        OID
   memberoid;
        returnMessage*
   giveNextReply();
        returnMessage*
   fcrossInvoke(const invokeMessage*,
            const segmentDesc** = 0,
            const OID& = nullOID);
 public:
      sosFamily (const OID&, const char*);
      sosFamily (const char*);
};
```

SYNOPSIS

```
#include <sos/context.h>
#include <sos/rpc.h>
sosFamily* membaddr = new sosFamily(familyName);
OID thisMemberOid;
sosFamily* membaddr = new sosFamily(thisMemberOid,familyName);
```

ARGUMENTS TO THE CONSTRUCTORS

familyName is the symbolic name of the family that this object is intended to be a member. This must be the same as the one given at the family's creation call.

thisMemberOid is the member identifier within its family.

DESCRIPTION

The sosFamily is the base class of any family's member. This basic object provides the common procedures of any member of any family. This basic class ensures the installation of the family's proxy (familyStub) and provides transparent accesses to this proxy.

The *memberOid* is the unique identifier of this member within its family.

One may creates a member with a specified *memberOid* or not.

If the *memberOid* is not supllied, the *memberOid* is equal to the *concreteOID* of this member.

SEE ALSO

creatFamily(6),fcrossInvoke(6),giveNextReply(6)

NAME

ASmain – the acquaintance service

SYNOPSIS

If file **predefContexts contains at least the line:**

ASmain 1

then running **sos** automatically starts the acquaintance service.

ASmain can also be started from the Unix shell by running **sos** in background, waiting for it to initialize, and then running **ASmain.**

DESCRIPTION

This is the binary file for the main acquaintance server on a machine.

FILES

~sos/v4/etc/predefContexts

SEE ALSO

intro(1), sos(7), acquaintanceService(3)

NAME

CSmain – the name service

SYNOPSIS

If file **predefContexts** contains the lines:

ASmain 1
SSmain 2
NSmain 3
CSmain 4

then running **sos** automatically starts the communication service.

CSmain can also be started from the Unix shell by running **sos, ASmain, SSmain** and **NSmain** in background, waiting for them to initialize, and then running **CSmain.**

DESCRIPTION

This is the binary file for the main communication server on a machine.

FILES

~sos/v4/etc/predefContexts

SEE ALSO

intro(1), sos(7)

NAME

NSmain – the name service

SYNOPSIS

If file **predefContexts** contains at least the lines:

ASmain 1
SSmain 2
NSmain 3

then running **sos** automatically starts the name service.

NSmain can also be started from the Unix shell by running **sos, ASmain,** and **SSmain** in background, waiting for them to initialize, and then running **NSmain.**

DESCRIPTION

This is the binary file for the main name server on a machine.

FILES

~sos/v4/etc/predefContexts

SEE ALSO

intro(1), sos(7), nameService(5)

NAME

SSmain – the storage service

SYNOPSIS

If file **predefContexts** contains at least the lines:

ASmain 1
SSmain 2

then running **sos** automatically starts the storage service.

SSmain can also be started from the Unix shell by running **sos** and **ASmain** in background, waiting for them to initialize, and then running **SSmain.**

DESCRIPTION

This is the binary file for the main storage server on a machine.

FILES

~sos/v4/etc/predefContexts

SEE ALSO

intro(1), sos(7), storageService(4)

NAME

makecode - creation of SOS code objects.

SYNOPSIS

makecode *className exportFile [-s symbolic-codeName] [-p symbolic-baseName]*

DESCRIPTION

Makecode creates a code object for the dynamic class *className*. This code object is stored on disk and registered under the directory service. The *exportFile* argument is the name of the **.o** file created by a previous **sosCC** compilation with the option **-c.**

The name of the resulting code object is, by default, the name of the class post-fixed by ".code" in the SOS "/export" directory object.

Makecode interprets the following options:

-s to specify a symbolic name *symbolic-codeName*. It is necessary if one wants to create more than one code object for a same dynamic class.

-p to specify a prerequisite of the code object, typically a base class *symbolic-baseName*

EXAMPLE

Suppose you have produce a file "A.o", for a dynamic class A. You can now run the command (under SOS):

makecode A A.o /export/A.code,

so that a principal able to export proxies of such a class will just have to write :

```
ref proxyCode;
proxy = new A;
NS->lookup( "/export/A.code", proxyCode );
proxy->setCodeRef( proxyCode );
```

FEATURES

makecode is a SOS utility which runs only under SOS.

DIAGNOSTICS

An error is reported if the object code can't be created properly or if the symbolic codeName (default or specific) already exists.

SEE ALSO

sosCC(1), code(2), nameService(5), acquaintanceService(3).

NAME

sos – start SOS

SYNOPSIS

sos

DESCRIPTION

Sos starts an SOS prototype under the UNIX operating system. Sos initiates the SOS kernel and then starts the contexts described by the **predefContexts** file. More contexts can be started as UNIX processes, e.g. from the shell (see *intro(1)*).

There is two ways to abort a context execution, either by executing the Unix **exit** instruction, or by sending SIGINT (e.g. by typing the interrupt character at the keyboard).

When sending the SIGTERM signal to **sos**, all active contexts are aborted and sos exits.

FILES

~sos/v4/etc/predefContexts

must exist. It has one line per context to run automatically at start-up. A line has the following structure:

contextName number,

where *contextName* is the name of binary file of context (compiled by **sosCC -o ...**), and *number*, if present, is used to generate a (unique) context OID when the context is started by *sos*. **predefContexts** file should contain at least the following line:

ASmain 1

(see *ASmain(7)*).

BUGS

It is possible to have only one SOS running on one machine at one time.

The number of contexts which may be started on one machine at one time is limited by the number of open Unix file descriptors.

SEE ALSO

intro(1), sosCC(1), ASmain(7), SSmain(7), NSmain(7), CSmain(7)

NAME

speak – on-line communication with other users

SYNOPSIS

speak

DESCRIPTION

speak implements a distributed on-line conference.

Any user can join a conference to exchange messages. **speak** is invoked without arguments. It is ready to accept commands when it gives you the list of users and prints its prompt ("speak>").

At this stage, if you wish to send a message to someone else, use the following format:

speak> *user* : the message

user is of the form:
logname [@hostname][_ttyname]

logname is the login name of a user who has joined the conference, and specifies the receiver of the message.
If you want to speak to a user who is recorded more than once, the *hostname* or the *ttyname* arguments may be used to discriminate the receiver.
A empty *logname* means that the message is broadcasted to all connected users.

BUGS

(1) The server **speakServer** must be active.

(2) To exit, just type ^C. This terminates the whole SOS and all running contexts.

(3) Only 8 users per host.

SEE ALSO

intro(1), sos(7), speakServer(7)

NAME

speakServer – the server of speak

SYNOPSIS

speakServer

DESCRIPTION

speakServer is a server used by the **speak** program.

BUGS

It is possible to have only one conference active at one time

SEE ALSO

speak(7)

NAME

DL_INFO – introduction to dynamic linking debug

SYNOPSIS

```
class DL_INFO{
  public:
    void* searchSymbol
          (char* name, int where= ALL_FILES);
    void* searchSymbol
          (char* name,
              char* filename, int where= ALL_FILES);
    char* searchSymbol
          (void* name, int where= ALL_FILES);

    int   searchAllOccurrences
          (char* name, SYMB_OCCUR* tbl,
              int noccur= 0, int where= ALL_FILES);

    SYMB_OCCUR* getSymbOccur();
    char*       symbolName();
    char        symbolType();
    char*       symbolTypeByString();
    void*       symbolAddress();
    char*       symbolFilename();

    int         printLoadedFilenames (FILE* = stderr);

#ifdef sun
    int printTable (FILE* = stderr);
    int printAllTable (FILE* = stderr);
    int printTable
        (char* namefile,
            FILE* = stderr, int where= ALL_FILES,
                int absolute= 0,int index= 0);
#endif /* sun */

    inline KEYTYPE classKey (char* class_name);
    inline KEYTYPE baseClassKey (char* class_name);

    inline void setUnderscore();
    inline void resetUnderscore();
    inline char isUnderscoreSet();

    void disableCache();
    DL_INFO();
    ~DL_INFO();

  protected:
    inline void setValid();
    inline void setInvalid();
    inline int  isValid();
};
```

DESCRIPTION

Dynamic linking informations are searched either in library files *(LIBRARIES_ONLY)*, or in user files *(USERFILES_ONLY)*, or in both *(ALL_FILES)*. The argument *where* in the declaration selects this research and is set to *ALL_FILES* by default.
Printing procedures go default output to *stderr*.
All addresses are absolute addresses.
It is noticed that all symbol or table researches use an identical algorithm.

1) Research symbol methods:

- **void* searchSymbol**
  (char* *name*, int *where*= ALL_FILES);
  Searches the symbol *name* in files list *where*.
  Returns address of the symbol or NULL if not found.

- **void* searchSymbol**
  (char* *name*, char* *filename*, int *where*= ALL_FILES);
  Searches the symbol *name* in file *filename* included in list *where*.
  Returns address of the symbol or NULL if not found.

- **char* searchSymbol**
  (void* *name*, int *where*= ALL_FILES);
  Searches the symbol at address *address* in files list *where*.
  Returns name of the symbol or NULL if not found.

- **int searchAllOccurrences**
  (char* *name*, SYMB_OCCUR* *tbl*,
  int *noccur*, int *where*= ALL_FILES);
  Searches the first *noccur* occurrences of *name* in files list *where* and put addresses in *tbl*. Puts 0 at the end of tbl, assuming user declaration:
  SYMB_OCCUR tbl[NOCCUR+1];

  +1 is needed to assign the last index to 0.
  Returns the exact number of occurrences.

2) Information methods:

Current instance of *DL_INFO* holds information on the last symbol reached by a previous call to any *searchSymbol* method described above (except *searchAllOccurrences)*. Methods described below allows to find information about this symbol.

- **SYMB_OCCUR* getSymbolOccur();**
  Returns a pointer to the symbol (see *SYMB_OCCUR(8))*, or NULL if information is not valid.

- **char* symbolName();**
  Returns the symbol name or the string "?" if information is not valid.

- **char symbolType();**
  Returns the symbol type (see *a.out(5))* or 0 if information is not valid.

- **char* symbolTypeByString();**
  Returns the symbol type (see the structure *nlist* in *a.out(5))* as string or the string "?" if

information is not valid.

- **void symbolAddress();**
  Returns the symbol address or NULL if information is not valid.

- **char* symbolFileName();**
  Returns the symbol file name or the string "?" if information is not valid.

3) Printing methods:

- **void printLoadedFilenames (FILE*);**
  Prints filenames of all loaded files.

- **void printAllTable (FILE*);**
  Prints symbol table for all loaded files.

- **int printTable (FILE*);**
  Prints symbol table of the file with respect to the last symbol reached by a previous call to *searchSymbol.*
  Returns 1 if information is valid, 0 otherwise.

- **int printTable**
  (char* *namefile*, FILE*, int *where*,
  int *absolute*, int *index*);
  Prints symbol table of file *namefile* in file list *where*. Filenames can be in a relative or absolute path. If argument *absolute* is set to the default value 0, the filename then considered is extracted from the path. If *absolute* is set to a non-zero value, the *filename* argument is use in a raw way.
  Different files can be loaded with an identical filename. The argument *index* selects the occurrence searched (from the default value 0).

4) Dynamic keys information:

- **KEYTYPE classKey (char* classname);**
  Returns the dynamic key of *classname.*
  Returns KEY_NOT_FOUND for any failure detected. If the request successes, the return key can be printed as long.

- **KEYTYPE baseClassKey (char* classname);**
  Returns the dynamic key of the (first) base class of *classname.*
  Returns KEY_NOT_FOUND for any failure detected. If the request successes, the return key can be printed as long.

5) Advanced routines:

In-core method names starts with the character '_'. By default, this character is hidden when any method defined above is invoking. Methods below are provided to set or not this character.

- **void setUnderscore();**
  Specifies that the application adds itself the character '_' in front of symbol names and then disables the automatic adding.

- **void resetUnderscore();**
  Specifies that the application does not add the character '_' in front of symbol names and then enables the automatic adding.

- **char isUnderscoreSet();**
  current status. Return VALID (set) or UNVALID (reset).

A cache is used to speed up symbol research.

- **disableCache();**
  Allows to disable use of cache. The enableCache method is not supplied.

- **DL_INFO();**
  This constructor sets internal flags and creates the cache.

- **~DL_INFO();**
  This destructor deletes the cache.

Internal datum allows to valid (or invalid) information about the symbol reached by a previous call to *searchSymbol*.

- **void setValid();**
  Current information is set VALID.

- **void setInvalid();**
  Current information is set INVALID.

- **int isValid();**
  Returns current status (VALID or INVALID).

EXAMPLE

The following procedure shows how to use the *DL_INFO* information.

```
#include "dyninfo.h"

void search_address (char* s){
   DL_INFO dlInfo;
   void* address;

   address = dlInfo.searchSymbol (s, LIBRARIES_ONLY);

   if (address != NULL){
      printf ("symb %s: add 0x%x in library file %s\n",
               s, address, dlInfo.symbolFilename());
      return;
   }

   address = dlInfo.searchSymbol (s, USERFILES_ONLY);
```

```
        if (address != NULL){
            printf ("symb %s: add 0x%x in user file %s\n",
                        s, address, dlInfo.symbolFilename());
            return;
        }

        printf ("symb %s: not found\n", s);
    }
```

SEE ALSO

dynamic(1), SYMB_OCCUR(5), dkeyof(5).
On the use of the dynamic link editor in SOS V4, in ~sos/v4/doc/dlink

BUGS

More debugging support is in progress.

NAME

SYMB_OCCUR – introduction to dynamic linking debug

SYNOPSIS

```
class SYMB_OCCUR {
 public:
  inline char* name();
  inline char* filename();
  inline void* address();
  inline char  type();
  char         typeByString();

  inline SYMB_OCCUR* iter();
  inline SYMB_OCCUR* next();
  inline int         last();

};
```

DESCRIPTION

SYMB_OCCUR stands for symbol occurrence. This class is supplied as complement to the class *DL_INFO* (see *DL_INFO(8)*). Each instance of this last class can hold or fill in information of instances of the class *SYMB_OCCUR*.

1) Information methods:

- **char* name();**
  Returns current symbol name.

- **char* filename();**
  Returns current symbol filename.

- **void* address();**
  Returns current symbol address.

- **char type();**
  Returns current symbol type (see the structure *nlist* in *a.out(5)*).

- **char* typebyString();**
  Returns current symbol type (see the structure *nlist* in *a.out(5)*) as string or the string "?" if dummy type.

2) Iterative methods:

Call to *DL_INFO::searchAllOccurrences* fills the following table *tbl:*

```
SYMB_OCCUR tbl[ NOCCUR+1 ];
char* symbol = ... ;
DL_INFO dlInfo;

dlInfo.searchAllOccurrences( symbol, tbl, NOCCUR );
```

The following methods provid an iterative way to lookup the table:

- **SYMB_OCCUR* iter();**
  Initializes the iterator.
  Returns initializer.

- **SYMB_OCCUR* next();**
  Returns the next element of the table, 0 if the last one.

- **int last();**
  Tests if the table lookup is over.
  Beware: return 0 if IS the last.

Assuming previous declarations and call, these routines can be exploited as follows:

```
for( SYMB_OCCUR* symb = tbl->iter ;
        symb->last() ;
            symb = symb->next())
{
   ...
}
```

EXAMPLE

A simple symbol search looks like this:

```
#include "dyninfo.h"

void search_symbol( char* symb ){
   DL_INFO dlInfo;

   if( dlInfo.searchSymbol (symb) == NULL ) return;

   SYMB_OCCUR* symb_occur = dlInfo.getSymbOccur();

   printf( "symbol %s: add 0x%x, type %s, filename %s0,
        symb_occur->name(),
            symb_occur->address(),
                symb_occur->typeByString(),
                    symb_occur->filename() );
}
```

SEE ALSO

dynamic(2), DL_INFO(8), dkeyof(2).
On the use of the dynamic link editor in SOS V4, in ~sos/v4/doc/dlink

NAME

Zdebug – a little bit of help for debugging dynamic classes.

SYNOPSIS

void Zdebug (FILE* = stderr);

DESCRIPTION

Zdebug provides a little help for for debugging dynamic classes.

It can be used to verify correct loading of dynamic classes, loaded either dynamically or statically.

For example, it can be called to verify static linking of dynamic classes, before any statements into the body of the main procedure, as in:

```
#include "dynload.h"

main(){
   Zdebug();

   ...
}
```

Output is redirected by default into the standard error (file descriptor 2).

EXAMPLE

We consider the following classes:

```
dynamic class A {
   int i;
   importRequest* ir;
   A( int );
   A( importRequest* );
   virtual m();
};

dynamic class B : public A {
   A a;
   B( int );
   B( importRequest* );
   virtual m();
};
```

where all methods but the second B constructor and B::m() are defined. We assume that they are loaded before calling to *Zdebug*.

- First, the following message appears:

***** DEBUG LINK OF DYNAMIC CODE *****

- Second, the class names are printed in the order of loading (both statically and dynamically).

A
B

- Third, each class is detailed:

```
** class A, dkey=-1700564103 nfunc=6, all_def=(cl 1, gl 0) **
     index:  address:                name:

     > -2    159742  (0x26ffe)       __A__ctorFPCimportRequest___
     > -1    17572   (0x44a4)        _sosDynError
     > 0     159830  (0x27056)       __A_m
     > 1     159646  (0x26f9e)       __A__ctorFI__
     > 2     159742  (0x26ffe)       __A__ctorFPCimportRequest___
     > 3     159830  (0x27056)       __A_m
     in table from: example.o

** class B, dkey=2006171841 nfunc=7, all_def=(cl 0, gl 0) **
     index:  address:                name:

     > -2    159742  (0x26ffe)       __A__ctorFPCimportRequest___
     > -1    17572   (0x44a4)        _sosDynError
     > 0     159830  (0x27056)       __A_m
     > 1     159866  (0x2707a)       __B__ctorFI__
     > 2     159742  (0x26ffe)       __A__ctorFPCimportRequest___
     > 3     159830  (0x27056)       __A_m
     > 4     17572   (0x44a4)        _sosDynError
     in table from: example.o
```

After the classname, appear the dynamic key, the function number (included negative slots) and two hints. All_def stands for all defined with respect to the class (cl 1 if all methods with positive index are defined) and to the global state (gl 1 if all methods of all classes are defined). The last line prints the file name where the methods were found.

SEE ALSO

dynamic(2) DL_INFO(8)

BUGS

Methods are printed with a C name. *Zdebug* does not replace a real debugger. The name can seem strange.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique