# Table of Contents

SCSCP C Library ................................................. 1

1 SCSCP C Library Copying conditions ............ 2

2 Introduction to SCSCP C Library .............. 3

3 Installing SCSCP C Library ......................... 4

3.1 Installation on a Unix-like system (Linux, Mac OS X, BSD, cygwin, ...) .......................................................... 4

3.1.1 Other ‘make’ Targets ........................................ 4

3.2 Installation on Windows system ................. 5

4 Reporting bugs ............................................... 6

5 SCSCP C Library Basics ................................. 7

5.1 Headers and Libraries ....................................... 7

5.1.1 Compilation on a Unix-like system ............... 7

5.2 Compilation on a Windows system ............... 7

5.2 Thread safe .................................................. 8

6 C Interface .................................................. 9

6.1 Constants ................................................... 9

6.2 Types ....................................................... 9

6.2.1 SCSCP_socketserver ....................................... 9

6.2.2 SCSCP_socketclient ...................................... 9

6.2.3 SCSCP_incomingclient .................................. 10

6.2.4 SCSCP_status ............................................. 10

6.2.5 SCSCP_calloptions ...................................... 11

6.2.6 SCSCP_returnoptions ................................... 12

6.2.7 SCSCP_msgtype .......................................... 12

6.2.8 SCSCP_encodingtype .................................... 12

6.2.9 SCSCP_xmlnodeptr ....................................... 12

6.2.10 SCSCP_xmlattrptr ...................................... 13

6.2.11 SCSCP_io ................................................ 13

6.3 Server functions ........................................ 13

6.3.1 SCSCP_ss_init ............................................ 13

6.3.2 SCSCP_ss_clear .......................................... 14

6.3.3 SCSCP_ss_listen .......................................... 14

6.3.4 SCSCP_ss_getport ....................................... 14

6.3.5 SCSCP_ss_close ......................................... 14
<table>
<thead>
<tr>
<th>Section</th>
<th>Function Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.3.6</td>
<td>SCSCP_ss_acceptclient</td>
<td>14</td>
</tr>
<tr>
<td>6.3.7</td>
<td>SCSCP_ss_closeincoming</td>
<td>15</td>
</tr>
<tr>
<td>6.3.8</td>
<td>SCSCP_ss_callrecvstr</td>
<td>15</td>
</tr>
<tr>
<td>6.3.9</td>
<td>SCSCP_ss_callrecvheader</td>
<td>16</td>
</tr>
<tr>
<td>6.3.10</td>
<td>SCSCP_ss_getxmlnode</td>
<td>16</td>
</tr>
<tr>
<td>6.3.11</td>
<td>SCSCP_ss_getxmlnoderawstring</td>
<td>16</td>
</tr>
<tr>
<td>6.3.12</td>
<td>SCSCP_ss_sendterminatedstr</td>
<td>17</td>
</tr>
<tr>
<td>6.3.13</td>
<td>SCSCP_ss_sendcompletedstr</td>
<td>17</td>
</tr>
<tr>
<td>6.3.14</td>
<td>SCSCP_ss_sendcompletedhook</td>
<td>17</td>
</tr>
<tr>
<td>6.3.15</td>
<td>SCSCP_ss_informessagesend</td>
<td>18</td>
</tr>
<tr>
<td>6.3.16</td>
<td>SCSCP_ss_set_encodingtype</td>
<td>18</td>
</tr>
<tr>
<td>6.3.17</td>
<td>SCSCP_ss_get_encodingtype</td>
<td>18</td>
</tr>
<tr>
<td>6.4</td>
<td>Client functions</td>
<td>18</td>
</tr>
<tr>
<td>6.4.1</td>
<td>SCSCP_sc_init</td>
<td>18</td>
</tr>
<tr>
<td>6.4.2</td>
<td>SCSCP_sc_clear</td>
<td>19</td>
</tr>
<tr>
<td>6.4.3</td>
<td>SCSCP_sc_connect</td>
<td>19</td>
</tr>
<tr>
<td>6.4.4</td>
<td>SCSCP_sc_close</td>
<td>19</td>
</tr>
<tr>
<td>6.4.5</td>
<td>SCSCP_sc_set_encodingtype</td>
<td>20</td>
</tr>
<tr>
<td>6.4.6</td>
<td>SCSCP_sc_get_encodingtype</td>
<td>20</td>
</tr>
<tr>
<td>6.4.7</td>
<td>SCSCP_sc_callsendstr</td>
<td>20</td>
</tr>
<tr>
<td>6.4.8</td>
<td>SCSCP_sc_callrecvstr</td>
<td>21</td>
</tr>
<tr>
<td>6.4.9</td>
<td>SCSCP_sc_callsendhook</td>
<td>21</td>
</tr>
<tr>
<td>6.4.10</td>
<td>SCSCP_sc_callrecvheader</td>
<td>21</td>
</tr>
<tr>
<td>6.4.11</td>
<td>SCSCP_sc_callrecvterminated</td>
<td>22</td>
</tr>
<tr>
<td>6.4.12</td>
<td>SCSCP_sc_callrecvcompleted</td>
<td>22</td>
</tr>
<tr>
<td>6.4.13</td>
<td>SCSCP_sc_callsendinterrupt</td>
<td>22</td>
</tr>
<tr>
<td>6.4.14</td>
<td>SCSCP_sc_executehookxmlnode</td>
<td>23</td>
</tr>
<tr>
<td>6.4.15</td>
<td>SCSCP_sc_executehookstr</td>
<td>23</td>
</tr>
<tr>
<td>6.4.16</td>
<td>SCSCP_sc_informessagesend</td>
<td>24</td>
</tr>
<tr>
<td>6.4.17</td>
<td>SCSCP_sc_getxmlnode</td>
<td>24</td>
</tr>
<tr>
<td>6.4.18</td>
<td>SCSCP_sc_getxmlnoderawstring</td>
<td>24</td>
</tr>
<tr>
<td>6.5</td>
<td>I/O functions</td>
<td>24</td>
</tr>
<tr>
<td>6.5.1</td>
<td>SCSCP_io_write</td>
<td>24</td>
</tr>
<tr>
<td>6.5.2</td>
<td>SCSCP_io_writeOMSTR</td>
<td>25</td>
</tr>
<tr>
<td>6.5.3</td>
<td>SCSCP_io_writeOMFdouble</td>
<td>25</td>
</tr>
<tr>
<td>6.5.4</td>
<td>SCSCP_io_writeOMFstr</td>
<td>25</td>
</tr>
<tr>
<td>6.5.5</td>
<td>SCSCP_io_writeOMInt</td>
<td>25</td>
</tr>
<tr>
<td>6.5.6</td>
<td>SCSCP_io_writeOMIlonglong</td>
<td>26</td>
</tr>
<tr>
<td>6.5.7</td>
<td>SCSCP_io_writeOMIstr</td>
<td>26</td>
</tr>
<tr>
<td>6.5.8</td>
<td>SCSCP_io_writeOMS</td>
<td>26</td>
</tr>
<tr>
<td>6.5.9</td>
<td>SCSCP_io_writeOMV</td>
<td>27</td>
</tr>
<tr>
<td>6.5.10</td>
<td>SCSCP_io_writeOMR</td>
<td>27</td>
</tr>
<tr>
<td>6.5.11</td>
<td>SCSCP_io_writeOMB</td>
<td>27</td>
</tr>
<tr>
<td>6.5.12</td>
<td>SCSCP_io_writebeginOMA</td>
<td>27</td>
</tr>
<tr>
<td>6.5.13</td>
<td>SCSCP_io_writeendOMA</td>
<td>28</td>
</tr>
<tr>
<td>6.5.14</td>
<td>SCSCP_io_writebeginOMATP</td>
<td>28</td>
</tr>
<tr>
<td>6.5.15</td>
<td>SCSCP_io_writeendOMATP</td>
<td>28</td>
</tr>
<tr>
<td>6.5.16</td>
<td>SCSCP_io_writebeginOMATTR</td>
<td>28</td>
</tr>
</tbody>
</table>
6.6 Status functions ............................................. 31
  6.6.1 SCSCP_status_clear .................................. 31
  6.6.2 SCSCP_status_is ...................................... 31
  6.6.3 SCSCP_status_strerror ................................. 31

6.7 Procedure call options functions ............................ 32
  6.7.1 SCSCP_co_init ........................................ 32
  6.7.2 SCSCP_co_clear ........................................ 32
  6.7.3 SCSCP_co_set_callid ................................. 32
  6.7.4 SCSCP_co_get_callid ................................. 33
  6.7.5 SCSCP_co_set_runtimelimit ........................... 33
  6.7.6 SCSCP_co_get_runtimelimit ........................... 33
  6.7.7 SCSCP_co_set_minmemory .............................. 33
  6.7.8 SCSCP_co_get_minmemory .............................. 34
  6.7.9 SCSCP_co_set_maxmemory .............................. 34
  6.7.10 SCSCP_co_get_maxmemory ........................... 34
  6.7.11 SCSCP_co_set_debuglevel ......................... 35
  6.7.12 SCSCP_co_get_debuglevel ......................... 35
  6.7.13 SCSCP_co_set_returntype ......................... 35
  6.7.14 SCSCP_co_get_returntype ......................... 35
  6.7.15 SCSCP_co_get_encodingtype ....................... 36

6.8 Procedure return options functions ........................ 36
  6.8.1 SCSCP_ro_init ..................................... 36
  6.8.2 SCSCP_ro_clear ..................................... 36
  6.8.3 SCSCP_ro_set_callid ................................ 36
  6.8.4 SCSCP_ro_get_callid ................................ 37
  6.8.5 SCSCP_ro_set_runtime ............................. 37
  6.8.6 SCSCP_ro_get_runtime ............................. 37
  6.8.7 SCSCP_ro_set_memory ............................. 37
  6.8.8 SCSCP_ro_get_memory ............................. 38
  6.8.9 SCSCP_ro_set_message ........................... 38
  6.8.10 SCSCP_ro_get_message ......................... 38

6.9 XML parsing functions .................................... 38
  6.9.1 SCSCP_xmlnode_getnext ......................... 39
  6.9.2 SCSCP_xmlnode_getname ......................... 39
  6.9.3 SCSCP_xmlnode_getid ............................ 39
  6.9.4 SCSCP_xmlnode_getchild ......................... 40
  6.9.5 SCSCP_xmlnode_getcontent .................... 40
  6.9.6 SCSCP_xmlnode_getattr .......................... 40
  6.9.7 SCSCP_xmlattr_getnext ....................... 40
6.9.8 SCSCP_xmlattr_getvalue .............................. 40
6.9.9 SCSCP_xmlnode_readOMS .............................. 40
6.9.10 SCSCP_xmlnode_readOMIstr ........................ 40
6.9.11 SCSCP_xmlnode_readOMIint .......................... 41
6.9.12 SCSCP_xmlnode_readOMFstr .......................... 41
6.9.13 SCSCP_xmlnode_readOMFdouble ........................ 41
6.9.14 SCSCP_xmlnode_readOMIstr .......................... 41
6.9.15 SCSCP_xmlnode_readOMR .............................. 41
6.9.16 SCSCP_xmlnode_readOMV .............................. 41
6.9.17 SCSCP_xmlnode_readpairOMSOMSTR ..................... 42

6.10 Remote objects functions .............................. 42
6.10.1 SCSCP_sc_re怎能objectstoresessionhook ............... 42
6.10.2 SCSCP_sc_re怎能objectstorepersistenthook .......... 42
6.10.3 SCSCP_sc_re怎能objectretrievexmlnode ................ 43
6.10.4 SCSCP_sc_re怎能objectretrievestr .................... 43
6.10.5 SCSCP_sc_re怎能objectunbind ........................ 44

6.11 Miscellaneous functions .............................. 44
6.11.1 SCSCP_get_libversionstr ............................ 44
6.11.2 SCSCP_get_libversionnum ............................ 44
6.11.3 SCSCP_CHECK_LIBVERSION ............................. 44

7 Design a SCSCP C server .............................. 45

8 Design a SCSCP C client .............................. 47

9 C++ Interface ........................................ 49
9.1 Server ............................................ 49
9.1.1 Constructor ..................................... 49
9.1.2 listen .......................................... 49
9.1.3 getport ....................................... 49
9.1.4 close ......................................... 49
9.1.5 eof ........................................... 49
9.1.6 acceptclient ................................... 50
9.2 IncomingClient ..................................... 50
9.2.1 Destructor ..................................... 50
9.2.2 eof ........................................... 51
9.3 Client ............................................ 51
9.3.1 Constructor ..................................... 51
9.3.2 connect ....................................... 51
9.3.3 close ......................................... 51
9.3.4 eof ........................................... 51
9.4 Procedure call ...................................... 52
9.4.1 ProcedureCall .................................... 52
9.4.1.1 ProcedureCall Constructor ....................... 52
9.4.1.2 set_runtimelimit ................................ 52
9.4.1.3 get_runtimelimit ............................... 52
9.4.1.4 set_minmemory ............................... 52
SCSCP C Library
1 SCSCP C Library Copying conditions

gastineau@imcce.fr

This library is governed by the CeCILL-C or CeCILL license under French law and abiding by the rules of distribution of free software. You can use, modify and/or redistribute the software under the terms of the CeCILL-C license as circulated by CEA, CNRS and INRIA at the following URL "http://www.cecill.info".

As a counterpart to the access to the source code and rights to copy, modify and redistribute granted by the license, users are provided only with a limited warranty and the software’s author, the holder of the economic rights, and the successive licensors have only limited liability.

In this respect, the user’s attention is drawn to the risks associated with loading, using, modifying and/or developing or reproducing the software by the user in light of its specific status of free software, that may mean that it is complicated to manipulate, and that also therefore means that it is reserved for developers and experienced professionals having in-depth computer knowledge. Users are therefore encouraged to load and test the software’s suitability as regards their requirements in conditions enabling the security of their systems and/or data to be ensured and, more generally, to use and operate it in the same conditions as regards security.

The fact that you are presently reading this means that you have had knowledge of the CeCILL-C or CeCILL license and that you accept its terms.
Chapter 2: Introduction to SCSCP C Library

2 Introduction to SCSCP C Library

This library is an implementation of the Symbolic Computation Software Composibility Protocol (SCSCP). The current implementation is based on the specification version 1.3 (see Chapter 12 [References], page 73).

This library provides API to develop client applications to access Computer Algebra Systems which support that protocol. So these client applications will be referred as ‘SCSCP client’ in this documentation.

Computer Algebra Systems could use the API to provide services to other applications using this protocol. So these Computer Algebra Systems will be referred as ‘SCSCP server’ in this documentation.

This library provides a C and C++ API to allow to be used in C or C++ programs.
3 Installing SCSCP C Library

3.1 Installation on a Unix-like system (Linux, Mac OS X, BSD, cygwin, ...)

To build SCSCP C Library, you first have to install Libxml2 version 2.6 or later (see http://xmlsoft.org/) on your computer. You need C and C++ compilers, such as gcc and g++. And you need a standard Unix 'make' program, plus some other standard Unix utility programs.

Here are the steps needed to install the library on Unix systems:

1. 'tar xzf scscp-1.0.3.tar.gz'
2. 'cd scscp-1.0.3'
3. './configure'
   Running configure might take a while. While running, it prints some messages telling which features it is checking for.
   configure recognizes the following options to control how it operates.
   --help  
   -h      Print a summary of all of the options to configure, and exit.
   --prefix=dir  
   Use dir as the installation prefix. See the command make install for the installation names.
4. 'make'
   This compiles SCSCP C Library in the working directory.
5. 'make check'
   This will make sure SCSCP C Library was built correctly.
   If you get error messages, please report them to gastineau@imcce.fr (See Chapter 4 [Reporting bugs], page 6, for information on what to include in useful bug reports).
6. 'make install'
   This will copy the files scscp.h, scscpxx.h and scscptypes.h to the directory /usr/local/include, the files libscscp.a and libscscpxx.a to the directory /usr/local/lib, and the file scscp.info to the directory /usr/local/share/info (or if you passed the --prefix option to configure, using the prefix directory given as argument to --prefix instead of /usr/local). Note: you need write permissions on these directories.

3.1.1 Other 'make' Targets

There are some other useful make targets:

- 'scscp.info' or 'info'
  Create an info version of the manual, in scscp.info.
- 'scscp.pdf' or 'pdf'
  Create a PDF version of the manual, in scscp.pdf.
Chapter 3: Installing SCSCP C Library

3.2 Installation on Windows system

To build SCSCP C Library, you first have to install Libxml2 version 2.6 or later (see http://xmlsoft.org/) on your computer. You need C and C++ compilers and a Windows SDK. It has been successfully compiled with the Windows Server 2003 R2 Platform SDK, the Windows SDK of Vista, and the Windows Server 2008 Platform SDK.

Here are the steps needed to install the library on Windows systems:

1. Expand the file `scscp-1.0.3.tar.gz`

2. Execute the command `cmd.exe` from the menu ‘Start’ / ‘Execute...’
   This will open a console window

3. `cd 'dir\scscp-1.0.3'`
   Go to the directory dir where SCSCP C Library has been expanded.

4. `nmake /f makefile.vc XMLDIR=dir`
   This compiles SCSCP C Library in the working directory. Use dir as the installation directory of the libxml2 library.

5. `nmake /f makefile.vc XMLDIR=dir check`
   This will make sure SCSCP C Library was built correctly.
   If you get error messages, please report them to gastineau@imcce.fr (See Chapter 4 [Reporting bugs], page 6, for information on what to include in useful bug reports).

6. `nmake /f makefile.vc install DESTDIR=dir`
   This will copy the files scscp.h, scscpxx.h and scscptypes.h to the directory dir\include, the files scscp.lib and scscpxx.lib to the directory dir\lib, the file scscp.info and scscp.pdf to the directory dir\doc. Note: you need write permissions on these directories.
Chapter 4: Reporting bugs

4 Reporting bugs

If you think you have found a bug in the SCSCP C Library, first have a look on the SCSCP C Library web page http://www.imcce.fr/trip/scscp/, in which case you may find there a workaround for it. Otherwise, please investigate and report it. We have made this library available to you, and it is not to ask too much from you, to ask you to report the bugs that you find.

There are a few things you should think about when you put your bug report together. You have to send us a test case that makes it possible for us to reproduce the bug. Include instructions on how to run the test case.

You also have to explain what is wrong; if you get a crash, or if the results printed are incorrect and in that case, in what way.

Please include compiler version information in your bug report. This can be extracted using ‘cc -V’ on some machines, or, if you’re using gcc, ‘gcc -v’. Also, include the output from ‘uname -a’ and the SCSCP version.

Send your bug report to: gastineau@imcce.fr. If you think something in this manual is unclear, or downright incorrect, or if the language needs to be improved, please send a note to the same address.
5 SCSCP C Library Basics

5.1 Headers and Libraries

All C declarations needed to use C interface are collected in the include file `scscp.h`. It is designed to work with both C and C++ compilers. All C++ declarations needed to use C++ interface are collected in the include file `scscpxx.h`.

You should include that file in any C program using the SCSCP C Library:

```c
#include <scscp.h>
```

You should include that file in any C++ program using the SCSCP C Library:

```c
#include <scscpxx.h>
```

Note however that the SCSCP constants use `NULL`, the header file `stdio.h` must be included before.

```c
#include <stdio.h>
#include <scscp.h>
```

5.1.1 Compilation on a Unix-like system

All C programs using SCSCP must link against the `libscscp` library and the Libxml2 library. On Unix-like system this can be done with `-lscscp 'xml2-config --libs'`, for example

```bash
gcc myprogram.c -o myprogram -lscscp 'xml2-config --libs'
```

All C++ programs using SCSCP must link against the `libscscpxx` and `libscscp` libraries and the Libxml2 library. On Unix-like system this can be done with `-lscscpxx -lscscp 'xml2-config --libs'`, for example

```bash
g++ myprogram.cpp -o myprogram -lscscpxx -lscscp 'xml2-config --libs'
```

If SCSCP C Library has been installed to a non-standard location then it may be necessary to use `-I` and `-L` compiler options to point to the right directories, and some sort of run-time path for a shared library.

5.1.2 Compilation on a Windows system

All C programs using SCSCP must link against the `scscp.lib` library and the Libxml2 library. On Windows system this can be done with `scscp.lib libxml2_a.lib iconv.lib wsock32.lib ws2_32.lib`, for example

```bash
cl.exe /out:myprogram myprogram.c scscp.lib libxml2_a.lib iconv.lib wsock32.lib ws2_32.lib
```

All C++ programs using SCSCP must link against the `scscpxx.lib` and `scscp.lib` libraries and the Libxml2 library.

On Windows system this can be done with `scscpxx.lib scscp.lib libxml2_a.lib iconv.lib wsock32.lib ws2_32.lib`, for example
cl.exe /out:myprogram myprogram.cpp scscpxx.lib scscp.lib \  
  libxml2_a.lib iconv.lib wsock32.lib ws2_32.lib

If SCSCP C Library has been installed to a non-standard location then it may be necessary to use /I and /LIBPATH: compiler options to point to the right directories.

5.2 Thread safe

SCSCP C Library is reentrant and thread-safe with some exceptions:

1. It’s safe for two threads to read from the same SCSCP variable simultaneously, but it’s not safe for one to read while the another might be writing, nor for two threads to write simultaneously.

2. If the standard I/O functions such as send are not reentrant then the SCSCP I/O functions using them will not be reentrant either.
6 C Interface

6.1 Constants

SCSCP_VERSION_MAJOR
This integer constant defines the major revision of this library. It can be used to distinguish different releases of this library.

SCSCP_VERSION_MINOR
This integer constant defines the minor revision of this library. It can be used to distinguish different releases of this library.

SCSCP_VERSION_PATCH
This integer constant defines the patch level revision of this library. It can be used to distinguish different releases of this library.

```c
#if (SCSCP_VERSION_MAJOR>=2)
  || (SCSCP_VERSION_MAJOR>=1 && SCSCP_VERSION_MINOR>=3)
... #endif
```

SCSCP_PROTOCOL_VERSION_1_3
This string defines the version string for the SCSCP specification version 1.3.

SCSCP_PROTOCOL_VERSION_1_2
This string defines the version string for the SCSCP specification version 1.2.

SCSCP_PROTOCOL_DEFAULTPORT
This integer is the default value on which port should listen the SCSCP server. The value 26133 for this port has been assigned to SCSCP by the Internet Assigned Numbers Authority (IANA) in November 2007, see http://www.iana.org/assignments/port-numbers.

6.2 Types

6.2.1 SCSCP_socketserver

SCSCP_socketserver [Data type]
This type contains all information of the SCSCP server.

Before using any object of this type, the function SCSCP_ss_init must be called.

6.2.2 SCSCP_socketclient

SCSCP_socketclient [Data type]
This type contains all information of the SCSCP client about the connection through a socket to a SCSCP server.

Before using any object of this type, the function SCSCP_sc_init must be called.
6.2.3 SCSCP_incomingclient

SCSCP_incomingclient [Data type]
This type contains all information of an incoming connection accepted by a server.

6.2.4 SCSCP_status

SCSCP_status [Data type]
This type contains all information about errors. Before using any object of this type, the SCSCP_STATUS_INITIALIZER must be used to initialize the status object.

```
SCSCP_status status = SCSCP_STATUS_INITIALIZER;
...
```

Each function of the library updates an object of this type if an error occurs during the processing. The value SCSCP_STATUS_IGNORE could be used in order to ignore the returned value by these functions.

The possible values are

- ‘SCSCP_STATUS_OK’
  No error occurs.

- ‘SCSCP_STATUS_ERRNO’
  The variable errno is set to a system error. The value of errno specifies the error.

- ‘SCSCP_STATUS_EXECFAILED’
  The remote execution fails.

- ‘SCSCP_STATUS_NOMEM’
  Not enough memory

- ‘SCSCP_STATUS_OPENMATHNOTVALID’
  The OpenMath expression isn’t valid.

- ‘SCSCP_STATUS_RECVCANCEL’
  The interrupt message "<? scscp cancel ?>" was received.

- ‘SCSCP_STATUS_RECVQUIT’
  The quit message "<? scscp quit ?>" was received or the socket is closed before receiving this message.

- ‘SCSCP_STATUS_USAGEUNKNOWNDEBUGLEVEL’
  The debug level isn’t available in the procedure call message.

- ‘SCSCP_STATUS_USAGEUNKNOWNMEM’
  The memory usage isn’t available in the procedure return message.

- ‘SCSCP_STATUS_USAGEUNKNOWNMESSAGE’
  The information message isn’t available in the procedure return message.

- ‘SCSCP_STATUS_USAGEUNKNOWNMINMEMORY’
  The minimal memory isn’t available in the procedure call message.
The maximal memory isn’t available in the procedure call message.

The return type isn’t available in the procedure call message.

The runtime usage isn’t available in the procedure return message.

The runtime limit usage isn’t available in the procedure call message.

The version negotiation fails.

The following values indicate an invalid usage of the library

The call identifier isn’t defined in the options.

The object call options passed to the function is NULL.

The object client passed to the function is NULL.

The object return options passed to the function is NULL.

The return type isn’t defined in the options.

The object server passed to the function is NULL.

The object stream passed to the function is NULL.

6.2.5 SCSCP_calloptions

SCSCP_calloptions

This type contains all information about the options for a procedure call. The attribute of the procedure call could be set using the functions SCSCP_co_set_xxx. The attribute of the procedure call could be get using the functions SCSCP_co_get_xxx.

The value SCSCP_CALLOPTIONS_DEFAULT could be used in order to use the default procedure call options. In this case, a unique call identifier will be generated using the prefix libSCSCP: and the procedure call will return no value (SCSCP_option_return nothing will be used).

Before using any object of this type, the function SCSCP_co_init must be called.
6.2.6 SCSCP_returnoptions

SCSCP_returnoptions [Data type]
This type contains all information about the options for a procedure return. The attribute of the procedure return could be set using the functions SCSCP_ro_set_xxx. The attribute of the procedure return could be get using the functions SCSCP_ro_get_xxx.

The value SCSCP_RETURNOPTIONS_IGNORE could be used in order to ignore the returned value by the function SCSCP_sc_callrecvheader, SCSCP_sc_callrecvstr.

Before using any object of this type, the function SCSCP_ro_init must be called.

6.2.7 SCSCP_msgtype

SCSCP_msgtype [Data type]
This type defines the type of sent messages between the client and the server.

The available values are

'SCSCP_msgtype_ProcedureTerminated'
The message is a "Procedure terminated". It is defined by the symbol procedure_terminated of the OpenMath Content Dictionary scscp1.

'SCSCP_msgtype_ProcedureCompleted'
The message is a "Procedure completed". It is defined by the symbol procedure_completed of the OpenMath Content Dictionary scscp1.

'SCSCP_msgtype_ProcedureCall'
The message is a "Procedure call". It is defined by the symbol procedure_call of the OpenMath Content Dictionary scscp1.

'SCSCP_msgtype_Interrupt'
The message is a "Interrupt" signal. It is defined by the processing instruction "<? scscp terminate ?>".

6.2.8 SCSCP_encodingtype

SCSCP_encodingtype [Data type]
This type defines the encoding type of the sent OpenMath Objects between the client and the server.

The available values are

'SCSCP_encodingtype_XML'
The OpenMath objects are encoded using the XML encoding. It is the default encoding for all connections.

'SCSCP_encodingtype_Binary'
The OpenMath objects are encoded using the Binary encoding.

6.2.9 SCSCP_xmlnodeptr

SCSCP_xmlnodeptr [Data type]
This type defines a pointer to a node of a XML tree.
6.2.10 SCSCP_xmlattrptr

SCSCP_xmlattrptr [Data type]
This type defines a pointer to an attribute of a node of type SCSCP_xmlnodeptr
(noode of a XML tree).

6.2.11 SCSCP_io

SCSCP_io [Data type]
This type defines a pointer to a low-level Input/output stream.

6.3 Server functions

The following functions manage all operations on the SCSCP_socketserver and SCSCP_incomingclient objects.

6.3.1 SCSCP_ss_init

int SCSCP_ss_init [Library Function]
( SCSCP_socketserver* server, SCSCP_status* status , const char* servicename, const char* serviceversion, const char* serviceid, ...)

It initializes the internal structure of the object server. The variadic arguments should be of the type const char* and the last argument must be NULL. The variadic parameters define the allowed version of SCSCP protocol that could be negotiated with the SCSCP server.

The arguments servicename, serviceversion and serviceid are used as the value of the attribute service_name, service_version and service_id of the Connection Initiation Message.

The constants SCSCP_PROTOCOL_VERSION_x_x could be used for the variadic parameters.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If SCSCP_ss_init fails, the value of status is set to the corresponding error value. SCSCP_STATUS_IGNORE could be used for status in order to ignore the returned value.

The following example shows how to initialize the server supporting the scscp versions "1.3" and "1.001".

```c
SCSCP_status status;
SCSCP_server server;
int res;

res = SCSCP_ss_init(&server, &status, "MYCAS","1","myid",
                    SCSCP_PROTOCOL_VERSION_1_3,
                    "1.001", NULL);
```
6.3.2 SCSCP_ss_clear

```c
int SCSCP_ss_clear ( SCSCP_socketserver* server, SCSCP_status* status )
```

[Library Function]

It clears the internal structure of the object `server` and frees allocated memory for this object by the function SCSCP_ss_init.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If SCSCP_ss_clear fails, the value of `status` is set to the corresponding error value. SCSCP_STATUS_IGNORE could be used for `status` in order to ignore the returned value.

6.3.3 SCSCP_ss_listen

```c
int SCSCP_ss_listen ( SCSCP_socketserver* server, int port, int firstavailable, SCSCP_status* status )
```

[Library Function]

`server` creates an internal queue for the incoming connections and starts to listen on the port of "localhost". If the `port` isn’t available and `firstavailable` = 0, it fails. If the `port` isn’t available and `firstavailable` = 1, it retries with the next port until it finds an available port. If `port` is 0, an available random port is chosen.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If SCSCP_ss_listen fails, the value of `status` is set to the corresponding error value. SCSCP_STATUS_IGNORE could be used for `status` in order to ignore the returned value.

6.3.4 SCSCP_ss_getport

```c
int SCSCP_ss_getport ( SCSCP_socketserver* server )
```

[Library Function]

It returns the port from the `server` is listening. If the function `listen` failed or is not called before, then this function returns -1.

6.3.5 SCSCP_ss_close

```c
int SCSCP_ss_close ( SCSCP_socketserver* server, SCSCP_status* status )
```

[Library Function]

`server` terminates to listen for the incoming connections.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If SCSCP_ss_close fails, the value of `status` is set to the corresponding error value. SCSCP_STATUS_IGNORE could be used for `status` in order to ignore the returned value.

6.3.6 SCSCP_ss_acceptclient

```c
int SCSCP_ss_acceptclient ( SCSCP_socketserver* server, SCSCP_incomingclient* incomingclient, SCSCP_status* status )
```

[Library Function]

`server` extracts the first connection request on the queue of pending connections. If no pending connections are present on the queue, it blocks the caller until a connection is present.

After the Connection Initiation, the server returns, in the argument `incomingclient`, an object to manage future exchanged messages.

After the transactions, `incomingclient` must be closed and cleared with the function SCSCP_ss_closeincoming.
On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If SCSCP_ss_acceptclient fails, the value of status is set to the corresponding error value. SCSCP_STATUS_IGNORE could be used for status in order to ignore the returned value.

The following example shows how to implement the main loop of the SCSCP server.

```c
SCSCP_status status;
SCSCP_incomingclient incomingclient;
SCSCP_server server;

/*initialization of the server */
SCSCP_ss_init(&server, &status, "MYCAS","1","myid",
 SCSCP_PROTOCOL_VERSION_1_3,
 "1.001", NULL);

SCSCP_ss_listen(&server, SCSCP_PROTOCOL_DEFAULTPORT, &status);

while (SCSCP_ss_acceptclient(&server, &incomingclient, &status))
{
    ... process incoming message ...

    SCSCP_ss_closeincoming(&incomingclient, &status);
}

/* destroy the server */
SCSCP_ss_close(& server, &status);
SCSCP_ss_clear(&server, &status);
```

### 6.3.7 SCSCP_ss_closeincoming

**int SCSCP_ss_closeincoming ( SCSCP_incomingclient* incomingclient, SCSCP_status* status)**

It closes the connection with the client and clears the object incomingclient.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If SCSCP_ss_closeincoming fails, the value of status is set to the corresponding error value. SCSCP_STATUS_IGNORE could be used for status in order to ignore the returned value.

### 6.3.8 SCSCP_ss_callrecvstr

**int SCSCP_ss_callrecvstr ( SCSCP_incomingclient* incomingclient, SCSCP_calloptions* options, SCSCP_msgtype* msgtype, char* openmathbuffer, SCSCP_status* status)**

It waits for an incoming message. When a new message is available, then it reads the attribute, the type and the content of the message sent by the client incomingclient.

The call options options could be get using the functions SCSCP_co_get_xxx. options must be initialized before with the function SCSCP_co_init.
On exit, the argument msgtype must be SCSCP_msgtype_ProcedureCall or SCSCP_msgtype_Interrupt. The client sends only "Procedure Call" or "Interrupt" message. On exit, if the argument msgtype is SCSCP_msgtype_Interrupt, options contains the call identifier of the interrupted procedure call.

On exit, the argument openmathbuffer contains the content of the message sent by the client. This string must be freed by the system call free.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If SCSCP_ss_callrecvstr fails, the value of status is set to the corresponding error value. SCSCP_STATUS_IGNORE could be used for status in order to ignore the returned value.

6.3.9 SCSCP_ss_callrecvheader

int SCSCP_ss_callrecvheader ( SCSCP_incomingclient* incomingclient, SCSCP_calloptions* options, SCSCP_msgtype* msgtype, SCSCP_status* status) [Library Function]

It reads the attribute and type of the message sent by the client incomingclient.

The call options options could be get using the functions SCSCP_co_get_xxx. options must be initialized before with the function SCSCP_co_init.

On exit, the argument msgtype must be SCSCP_msgtype_ProcedureCall or SCSCP_msgtype_Interrupt. The client sends only "Procedure Call" or "Interrupt" message. On exit, if the argument msgtype is SCSCP_msgtype_Interrupt, options contains the call identifier of the interrupted procedure call.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If SCSCP_ss_callrecvheader fails, the value of status is set to the corresponding error value. SCSCP_STATUS_IGNORE could be used for status in order to ignore the returned value.

6.3.10 SCSCP_ss_getxmlnode

SCSCP_xmlnodeptr SCSCP_ss_getxmlnode ( SCSCP_incomingclient* incomingclient, SCSCP_status* status) [Library Function]

This function returns a pointer to the current XML tree received from the client. This function could be used to start parsing the message sent by the client. This pointer is valid until the next call to the functions SCSCP_ss_callrecv....

On exit, it returns NULL if an error occurs, otherwise the return value is a valid address. If SCSCP_ss_getxmlnode fails, the value of status is set to the corresponding error value. SCSCP_STATUS_IGNORE could be used for status in order to ignore the returned value.

6.3.11 SCSCP_ss_getxmlnoderastring

char * SCSCP_ss_getxmlnoderastring ( SCSCP_incomingclient* incomingclient, SCSCP_xmlnodeptr curnode, SCSCP_status* status) [Library Function]

This function returns the content of the current node curnode and its children as a string. This string must be freed by the system call free.

On exit, it returns NULL if an error occurs, otherwise the return value is a valid address. If SCSCP_ss_getxmlnoderastring fails, the value of status is set to the corresponding error value. SCSCP_STATUS_IGNORE could be used for status in order to ignore the returned value.
6.3.12 SCSCP_ss_sendterminatedstr

```c
int SCSCP_ss_sendterminatedstr ( SCSCP_incomingclient* incomingclient, SCSCP_returnoptions* options, const char * cdname, const char * symbolname, const char * message, SCSCP_status* status )
```

It sends a "procedure terminated" message to the SCSCP client with the options. The symbol of the OpenMath Error is defined by its name `symbolname` and its CD `cdname`. `message` is the message that will be inserted in a OMSTR OpenMath object.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If `SCSCP_ss_sendterminatedstr` fails, the value of `status` is set to the corresponding error value. `SCSCP_STATUS_IGNORE` could be used for `status` in order to ignore the returned value.

```c
SCSCP_ss_sendterminatedstr(&incomingclient, &options, "scscp1","error_system_specific", "can’t store a remote object", &status);
```

6.3.13 SCSCP_ss_sendcompletedstr

```c
int SCSCP_ss_sendcompletedstr ( SCSCP_incomingclient* incomingclient, SCSCP_returnoptions* options, const char * openmathbuffer, SCSCP_status* status )
```

It sends a "procedure completed" message to the SCSCP client with the options. The string `openmathbuffer` is the argument of the "procedure completed".

The string `openmathbuffer` must be a valid OpenMath command or NULL.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If `SCSCP_ss_sendcompletedstr` fails, the value of `status` is set to the corresponding error value. `SCSCP_STATUS_IGNORE` could be used for `status` in order to ignore the returned value.

6.3.14 SCSCP_ss_sendcompletedhook

```c
int SCSCP_ss_sendcompletedhook ( SCSCP_incomingclient* incomingclient, SCSCP_returnoptions* options, int (*callbackwriteargs)(SCSCP_io* stream, void *param, SCSCP_status* status), void* param, SCSCP_status* status )
```

It sends a "procedure completed" message to the client with the options. The arguments of the "procedure completed" message must be written by the callback function `callbackwriteargs`. The function `callbackwriteargs` must use the I/O functions `SCSCP_io_writexxx`, such as `SCSCP_io_writeOMSTR`, to write data which are sent to the SCSCP client.

The argument `param` is a pointer which is provided to the `callbackwriteargs` to exchange information. This pointer and its content isn’t modified by `SCSCP_ss_sendcompletedhook`.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If `SCSCP_ss_sendcompletedhook` fails, the value of `status` is set to the corresponding error value. `SCSCP_STATUS_IGNORE` could be used for `status` in order to ignore the returned value.
6.3.15 SCSCP_ss_infomessagesend

int SCSCP_ss_infomessagesend ( SCSCP_incomingclient* incomingclient, const char* messagebuffer, SCSCP_status* status )

It sends an information message to the SCSCP client for a debugging purpose. The string messagebuffer must be a valid string.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If SCSCP_ss_infomessagesend fails, the value of status is set to the corresponding error value. SCSCP_STATUS_IGNORE could be used for status in order to ignore the returned value.

6.3.16 SCSCP_ss_set_encodingtype

int SCSCP_ss_set_encodingtype ( SCSCP_incomingclient* incomingclient, SCSCP_encodingtype encodingtype, SCSCP_status* status )

This function sets the current encoding of the OpenMath objects used by the SCSCP server to send an answer. The provided Openmath buffers, such as for the call SCSCP_ss_sendcompletedstr, must use the same encoding.

The default encoding for the SCSCP server is the SCSCP_encodingtype_XML.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If SCSCP_ss_set_encodingtype fails, the value of status is set to the corresponding error value. SCSCP_STATUS_IGNORE could be used for status in order to ignore the returned value.

6.3.17 SCSCP_ss_get_encodingtype

int SCSCP_ss_get_encodingtype ( SCSCP_incomingclient* incomingclient, SCSCP_encodingtype* encodingtype, SCSCP_status* status )

This function returns, in the argument encodingtype, the current encoding for the OpenMath objects when the SCSCP server sends an answer.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If SCSCP_ss_get_encodingtype fails, the value of status is set to the corresponding error value. SCSCP_STATUS_IGNORE could be used for status in order to ignore the returned value.

6.4 Client functions

The following functions manage all operations on the SCSCP_socketclient object.

6.4.1 SCSCP_sc_init

int SCSCP_sc_init ( SCSCP_socketclient* client, SCSCP_status* status, ... )

It initializes the internal structure of the object client. The variadic arguments should be of the type const char* and the last argument must be NULL. The variadic parameters define the allowed version of SCSCP protocol that could be negotiated with the SCSCP server.
During the negotiation with the server, the client will choose the first version in the
variadic parameters that the server supports too. So the variadic parameters should start
by from the highest level of the SCSCP version to the lowest version.

The constants `SCSCP_PROTOCOL_VERSION_x_x` could be used for the variadic parameters.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If
`SCSCP_sc_init` fails, the value of `status` is set to the corresponding error value. `SCSCP_-
STATUS_IGNORE` could be used for `status` in order to ignore the returned value.

```c
res = SCSCP_sc_init(&client, &status, SCSCP_PROTOCOL_VERSION_1_3,
                   "1.0beta", NULL);
```

### 6.4.2 SCSCP_sc_clear

```c
int SCSCP_sc_clear ( SCSCP_socketclient* client, SCSCP_status* status )
                    [Library Function]
```

It clears the internal structure of the object `client` and frees allocated memory for this
object by the function `SCSCP_sc_init`. If a connection was already opened, the function
`SCSCP_sc_close` is called before clearing the object.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If
`SCSCP_sc_clear` fails, the value of `status` is set to the corresponding error value. `SCSCP_-
STATUS_IGNORE` could be used for `status` in order to ignore the returned value.

### 6.4.3 SCSCP_sc_connect

```c
int SCSCP_sc_connect ( SCSCP_socketclient* client, const char *machine, int port,
                      SCSCP_status* status )
                      [Library Function]
```

It tries to connect to the SCSCP server which is running on the computer `machine` and
is listening on the port `port`.

`client` must be initialized with the function `SCSCP_sc_init` before calling this function.
If the connection achieves, `client` is updated on exit.

`machine` could be any string but it must resolved as an IP address. Its value could be
"localhost" if the SCSCP server runs on the same computer.

In most of the case, the default value `SCSCP_PROTOCOL_DEFAULTPORT` should be
used for the port number.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If
`SCSCP_sc_connect` fails, the value of `status` is set to the corresponding error value. `SCSCP_-
STATUS_IGNORE` could be used for `status` in order to ignore the returned value.

### 6.4.4 SCSCP_sc_close

```c
int SCSCP_sc_close ( SCSCP_socketclient* client,
                     SCSCP_status* status )
                     [Library Function]
```

It closes a connection previously opened by a SCSCP client with the function `SCSCP_-
sc_connect`. 
On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If SCSCP_sc_close fails, the value of status is set to the corresponding error value. SCSCP_STATUS_IGNORE could be used for status in order to ignore the returned value.

### 6.4.5 SCSCP_sc_set_encodingtype

```c
int SCSCP_sc_set_encodingtype ( SCSCP_socketclient* client, SCSCP_encodingtype encodingtype, SCSCP_status* status )
```

This function sets the current encoding of the OpenMath objects used by the SCSCP client to send a "procedure call" message. The provided OpenMath buffers, such as for the call SCSCP_sc_callsendstr, must use the same encoding.

The default encoding for the SCSCP client is the SCSCP_encodingtype_XML.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If SCSCP_sc_set_encodingtype fails, the value of status is set to the corresponding error value. SCSCP_STATUS_IGNORE could be used for status in order to ignore the returned value.

### 6.4.6 SCSCP_sc_get_encodingtype

```c
int SCSCP_sc_set_encodingtype ( SCSCP_socketclient* client, SCSCP_encodingtype* encodingtype, SCSCP_status* status )
```

This function returns, in the argument encodingtype, the current encoding for the OpenMath objects when the SCSCP client sends a "procedure call" message.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If SCSCP_sc_get_encodingtype fails, the value of status is set to the corresponding error value. SCSCP_STATUS_IGNORE could be used for status in order to ignore the returned value.

### 6.4.7 SCSCP_sc_callsendstr

```c
int SCSCP_sc_callsendstr ( SCSCP_socketclient* client, SCSCP_calloptions* options, const char * openmathbuffer, SCSCP_status* status )
```

The client sends a "procedure call" message to the SCSCP server with the options. The string openmathbuffer is the argument of the procedure call. A connection must be previously opened with SCSCP_sc_connect before preforming this procedure call.

The string openmathbuffer must be an OpenMath Application object.

The value SCSCP_CALLOPTIONS_DEFAULT could be used for the parameter options in order to use the default procedure call options. The procedure call options could be set using the functions SCSCP_co_set_xxx.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If SCSCP_sc_callsendstr fails, the value of status is set to the corresponding error value. SCSCP_STATUS_IGNORE could be used for status in order to ignore the returned value.
6.4.8 SCSCP_sc_callrecvstr

```c
int SCSCP_sc_callrecvstr ( SCSCP_socketclient* client, [Library Function]
                      SCSCP_returnoptions* options, SCSCP_msgtype* msgtype, char**
                      openmathbuffer, SCSCP_status* status)
```

The `client` reads the attribute, the type and the content of the message returned by the server in response of a procedure call.

On exit, the argument `msgtype` contains the message type returned by the server. On exit, the argument `openmathbuffer` contains the content of the message returned by the server. This string must be freed by the system call `free`.

The value `SCSCP_RETURNOPTIONS_IGNORE` could be used for the parameter `options` in order to ignore the returned value. The return options could be get using the functions `SCSCP_ro_get_xxx`. If `options` isn’t `SCSCP_RETURNOPTIONS_IGNORE`, it must be initialized before with the function `SCSCP_ro_init`.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If `SCSCP_sc_callrecvstr` fails, the value of `status` is set to the corresponding error value. `SCSCP_STATUS_IGNORE` could be used for `status` in order to ignore the returned value.

6.4.9 SCSCP_sc_callsendhook

```c
int SCSCP_sc_callsendhook ( SCSCP_socketclient* client, [Library Function]
                           SCSCP_calloptions* options, int (*callbackwriteappli)(SCSCP_io*
                           stream, void *param, SCSCP_status* status), void* param, SCSCP_status*
                           status)
```

The `client` sends a "procedure call" message to the server with the `options`. The arguments of the procedure call must be written by the callback function `callbackwriteappli`. The function `callbackwriteappli` must use the functions `SCSCP_io_writexxx`, such as `SCSCP_io_writeOMSTR`, to write data which are sent to the SCSCP server. The function `callbackwriteappli` must write an OpenMath Application object. A connection must be previously opened with `SCSCP_sc_connect` before preforming this procedure call.

The argument `param` is a pointer which is provided to the `callbackwriteargs` to exchange information. This pointer and its content isn’t modified by `SCSCP_sc_callsendhook`.

The value `SCSCP_CALLOPTIONS_DEFAULT` could be used for the parameter `options` in order to use the default procedure call options. The procedure call options could be set using the functions `SCSCP_co_set_xxx`.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If `SCSCP_sc_callsendhook` fails, the value of `status` is set to the corresponding error value. `SCSCP_STATUS_IGNORE` could be used for `status` in order to ignore the returned value.

6.4.10 SCSCP_sc_callrecvheader

```c
int SCSCP_sc_callrecvheader ( SCSCP_socketclient* client, [Library Function]
                            SCSCP_returnoptions* options, SCSCP_msgtype* msgtype,
                            SCSCP_status* status)
```

The `client` reads the attribute and type of the message returned by the server in response of a procedure call.
The value SCSCP_RETURNOPTIONS_IGNORE could be used for the parameter options in order to ignore the returned value. The return options could be get using the functions SCSCP_ro_get_xxx. If options isn’t SCSCP_RETURNOPTIONS_IGNORE, it must be initialized before with the function SCSCP_ro_init.

On exit, the argument msgtype contains the message type returned by the server.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If SCSCP_sc_callrecvheader fails, the value of status is set to the corresponding error value. SCSCP_STATUS_IGNORE could be used for status in order to ignore the returned value.

6.4.11 SCSCP_sc_callrecvterminated

int SCSCP_sc_callrecvterminated ( SCSCP_socketclient* client, char ** cdname, char ** symbolname, char** messagebuffer, SCSCP_status* status)

The client reads the content of the OpenMath error if the server replies with a "procedure terminated" message. This function must be called only if SCSCP_sc_callrecvheader returns msgtype=SCSCP_msgtype_ProcedureTerminated.

On exit, the argument symbolname contains the name of the OpenMath symbol from the OpenMath content dictionary cdname. These strings must be freed by the system call free. On exit, the argument messagebuffer contains the full content of the error (including the head symbol of the OpenMath error) returned by the server. This string must be freed by the system call free.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If SCSCP_sc_callrecvterminated fails, the value of status is set to the corresponding error value. SCSCP_STATUS_IGNORE could be used for status in order to ignore the returned value.

6.4.12 SCSCP_sc_callrecvcompleted

int SCSCP_sc_callrecvcompleted ( SCSCP_socketclient* client, char** openmathbuffer, SCSCP_status* status)

The client reads the content of the messages if the server replies with a "procedure completed" message. This function must be called only if SCSCP_sc_callrecvheader returns msgtype=SCSCP_msgtype_ProcedureCompleted.

On exit, the argument openmathbuffer contains the content of the message returned by the server. This string must be freed by the system call free.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If SCSCP_sc_callrecvcompleted fails, the value of status is set to the corresponding error value. SCSCP_STATUS_IGNORE could be used for status in order to ignore the returned value.

6.4.13 SCSCP_sc_callsendinterrupt

int SCSCP_sc_callsendinterrupt ( SCSCP_socketclient* client, const char* call_id, SCSCP_status* status)

It sends an interrupt signal to the SCSCP server with the call ID call_id. The server need not to complete the computation but the server will always reply to the procedure call. The argument call_id can’t be NULL.
On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If SCSCP_sc_callsendinterrupt fails, the value of status is set to the corresponding error value. SCSCP_STATUS_IGNORE could be used for status in order to ignore the returned value.

6.4.14 SCSCP_sc_executehookxmlnode

```c
int SCSCP_sc_executehookxmlnode (SCSCP_socketclient* client, SCSCP_option_return returntype, const char * cdname, const char * symbolname, int (*callbackwriteargs)(SCSCP_io* stream, void *param, SCSCP_status* status), void* param, SCSCP_xmlnodeptr* node, SCSCP_status* status)
```

The client sends a "procedure call" message to the server with an OpenMath symbol `symbolname` from the content dictionary `cdname` as first argument. The other arguments of the procedure call must be written by the callback function `callbackwriteargs`. The function `callbackwriteargs` must use the functions SCSCP_io_writexxx, such as SCSCP_io_writeOMSTR, to write data which are sent to the SCSCP server. A connection must be previously opened with SCSCP_sc_connect before preforming this procedure call.

The argument `param` is a pointer which is provided to the `callbackwriteargs` to exchange information. This pointer and its content isn’t modified by SCSCP_sc_executehookxmlnode.

The server returns a result (object, cookie, nothing) depending on the value `returntype`. On exit, the argument `node` contains a pointer to the OpenMath object by the server.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If SCSCP_sc_executehookxmlnode fails, the value of `status` is set to the corresponding error value. SCSCP_STATUS_IGNORE could be used for `status` in order to ignore the returned value.

6.4.15 SCSCP_sc_executehookstr

```c
int SCSCP_sc_executehookstr (SCSCP_socketclient* client, SCSCP_option_return returntype, const char * cdname, const char * symbolname, int (*callbackwriteargs)(SCSCP_io* stream, void *param, SCSCP_status* status), void* param, char ** openmathbuffer, SCSCP_status* status)
```

The client sends a "procedure call" message to the server with an OpenMath symbol `symbolname` from the content dictionary `cdname` as first argument. The other arguments of the procedure call must be written by the callback function `callbackwriteargs`. The function `callbackwriteargs` must use the functions SCSCP_io_writexxx, such as SCSCP_io_writeOMSTR, to write data which are sent to the SCSCP server. A connection must be previously opened with SCSCP_sc_connect before preforming this procedure call.

The argument `param` is a pointer which is provided to the `callbackwriteargs` to exchange information. This pointer and its content isn’t modified by SCSCP_sc_callsendhookstr.

The server returns a result (object, cookie, nothing) depending on the value `returntype`. On exit, the argument `openmathbuffer` contains a string representing the OpenMath object by the server.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If SCSCP_sc_callsendhookstr fails, the value of `status` is set to the corresponding error value. SCSCP_STATUS_IGNORE could be used for `status` in order to ignore the returned value.
6.4.16 SCSCP_sc_infomessagesend

int SCSCP_sc_infomessagesend ( SCSCP_socketclient* client, const char* messagebuffer, SCSCP_status* status )

It sends an information message to the SCSCP server for a debugging purpose. The string messagebuffer must be a valid string.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If SCSCP_sc_infomessagesend fails, the value of status is set to the corresponding error value. SCSCP_STATUS_IGNORE could be used for status in order to ignore the returned value.

6.4.17 SCSCP_sc_getxmlnode

SCSCP_xmlnodeptr SCSCP_sc_getxmlnode ( SCSCP_socketclient* client, SCSCP_status* status )

The client returns a pointer to the current XML tree. This function could be used to start parsing the message sent by the server. This pointer is valid until the next call to the functions SCSCP_sc_callrecv....

On exit, it returns NULL if an error occurs, otherwise the return value is a valid address. If SCSCP_sc_getxmlnode fails, the value of status is set to the corresponding error value. SCSCP_STATUS_IGNORE could be used for status in order to ignore the returned value.

6.4.18 SCSCP_sc_getxmlnoderastring

char * SCSCP_sc_getxmlnoderastring ( SCSCP_socketclient* client, SCSCP_status* status )

This function returns the content of the current node curnode and its children as a string. This string must be freed by the system call free.

On exit, it returns NULL if an error occurs, otherwise the return value is a valid address. If SCSCP_sc_getxmlnoderastring fails, the value of status is set to the corresponding error value. SCSCP_STATUS_IGNORE could be used for status in order to ignore the returned value.

6.5 I/O functions

6.5.1 SCSCP_io_write

int SCSCP_io_write ( SCSCP_io* stream, const char* buffer, SCSCP_status* status )

It writes the data buffer directly to the stream. The argument buffer can’t be NULL.

This function must only be called by the callback function callbackwritearg provided to the hook functions SCSCP_ss_sendcompletedhook, SCSCP_sc_callsendhook, ....

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If SCSCP_io_write fails, the value of status is set to the corresponding error value. SCSCP_STATUS_IGNORE could be used for status in order to ignore the returned value.
6.5.2 SCSCP_io_writeOMSTR

int SCSCP_io_writeOMSTR ( SCSCP_io* stream, const char *buffer, const char *id, SCSCP_status* status )

It writes the data buffer as an OpenMath string <OMSTR>.../OMSTR> to the stream. If the argument buffer is NULL, the OpenMath string is encoding with only one space. id is the id of this object for the future reference (see OMR). id could be NULL if unset.

This function must only be called by the callback function callbackwritearg provided to the hook functions SCSCP_ss_sendcompletedhook, SCSCP_sc_callsendhook, ....

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If SCSCP_io_writeOMSTR fails, the value of status is set to the corresponding error value. SCSCP_STATUS_IGNORE could be used for status in order to ignore the returned value.

6.5.3 SCSCP_io_writeOMFdouble

int SCSCP_io_writeOMFdouble ( SCSCP_io* stream, double x, const char *id, SCSCP_status* status )

It writes the floating-point number x as an OpenMath float <OMF dec="..." /> to the stream. id is the id of this object for the future reference (see OMR). id could be NULL if unset.

This function must only be called by the callback function callbackwritearg provided to the hook functions SCSCP_ss_sendcompletedhook, SCSCP_sc_callsendhook, ....

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If SCSCP_io_writeOMFdouble fails, the value of status is set to the corresponding error value. SCSCP_STATUS_IGNORE could be used for status in order to ignore the returned value.

6.5.4 SCSCP_io_writeOMFstr

int SCSCP_io_writeOMFstr ( SCSCP_io* stream, const char* buffer, const char *id, SCSCP_status* status )

It writes the floating-point number buffer as an OpenMath float <OMF dec="..." /> to the stream. The argument buffer can’t be NULL. id is the id of this object for the future reference (see OMR). id could be NULL if unset.

This function must only be called by the callback function callbackwritearg provided to the hook functions SCSCP_ss_sendcompletedhook, SCSCP_sc_callsendhook, ....

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If SCSCP_io_writeOMFstr fails, the value of status is set to the corresponding error value. SCSCP_STATUS_IGNORE could be used for status in order to ignore the returned value.

6.5.5 SCSCP_io_writeOMIint

int SCSCP_io_writeOMIint ( SCSCP_io* stream, int x, const char *id, SCSCP_status* status )

It writes the integer x as an OpenMath integer <OMI>...</OMI> to the stream. id is the id of this object for the future reference (see OMR). id could be NULL if unset.

This function must only be called by the callback function callbackwritearg provided to the hook functions SCSCP_ss_sendcompletedhook, SCSCP_sc_callsendhook, ....
On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If SCSCP_io_writeOMIInt fails, the value of status is set to the corresponding error value. SCSCP_STATUS_IGNORE could be used for status in order to ignore the returned value.

### 6.5.6 SCSCP_io_writeOMIlonglong

```c
int SCSCP_io_writeOMIlonglong ( SCSCP_io* stream, long long x, const char *id, SCSCP_status* status )
```

It writes the integer \( x \) as an OpenMath integer \(<OMI>...</OMI>\) to the stream. \( id \) is the id of this object for the future reference (see OMR). \( id \) could be NULL if unset.

This function must only be called by the callback function `callbackwritearg` provided to the hook functions SCSCP_ss_sendcompletedhook, SCSCP_sc_callsendhook, ....

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If SCSCP_io_writeOMIlonglong fails, the value of status is set to the corresponding error value. SCSCP_STATUS_IGNORE could be used for status in order to ignore the returned value.

### 6.5.7 SCSCP_io_writeOMIstr

```c
int SCSCP_io_writeOMIstr ( SCSCP_io* stream, const char* buffer, const char *id, SCSCP_status* status )
```

It writes the integer \( buffer \) as an OpenMath integer \(<OMI>...</OMI>\) to the stream. The argument \( buffer \) can’t be NULL. \( id \) is the id of this object for the future reference (see OMR). \( id \) could be NULL if unset.

This function must only be called by the callback function `callbackwritearg` provided to the hook functions SCSCP_ss_sendcompletedhook, SCSCP_sc_callsendhook, ....

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If SCSCP_io_writeOMIstr fails, the value of status is set to the corresponding error value. SCSCP_STATUS_IGNORE could be used for status in order to ignore the returned value.

### 6.5.8 SCSCP_io_writeOMS

```c
int SCSCP_io_writeOMS ( SCSCP_io* stream, const char* cdname, const char* symbolname, const char *id, SCSCP_status* status )
```

It writes the symbol \( symbolname \) of the Content Dictionary \( cdname \) as an OpenMath symbol \(<OMS cd="..." name="..." />\) to the stream. The argument \( cdname \) and \( symbolname \) can’t be NULL. \( id \) is the id of this object for the future reference (see OMR). \( id \) could be NULL if unset.

This function must only be called by the callback function `callbackwritearg` provided to the hook functions SCSCP_ss_sendcompletedhook, SCSCP_sc_callsendhook, ....

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If SCSCP_io_writeOMS fails, the value of status is set to the corresponding error value. SCSCP_STATUS_IGNORE could be used for status in order to ignore the returned value.
6.5.9 SCSCP_io_writeOMV

```c
int SCSCP_io_writeOMV ( SCSCP_io* stream, const char* buffer, const char *id, SCSCP_status* status )
```

It writes the variable `buffer` as an OpenMath variable `<OMV name="..." />` to the `stream`. The argument `buffer` can't be NULL. `id` is the id of this object for the future reference (see OMR). `id` could be NULL if unset.

This function must only be called by the callback function `callbackwritearg` provided to the hook functions `SCSCP_ss_sendcompletedhook`, `SCSCP_sc_callsendhook`, ....

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value.

If `SCSCP_io_writeOMV` fails, the value of `status` is set to the corresponding error value. `SCSCP_STATUS_IGNORE` could be used for `status` in order to ignore the returned value.

6.5.10 SCSCP_io_writeOMR

```c
int SCSCP_io_writeOMR ( SCSCP_io* stream, const char* buffer, SCSCP_status* status )
```

It writes the reference `buffer` as an OpenMath reference `<OMR href="..."/>` to the `stream`. The argument `buffer` can't be NULL.

This function must only be called by the callback function `callbackwritearg` provided to the hook functions `SCSCP_ss_sendcompletedhook`, `SCSCP_sc_callsendhook`, ....

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value.

If `SCSCP_io_writeOMR` fails, the value of `status` is set to the corresponding error value. `SCSCP_STATUS_IGNORE` could be used for `status` in order to ignore the returned value.

6.5.11 SCSCP_io_writeOMB

```c
int SCSCP_io_writeOMB ( SCSCP_io* stream, const void* buffer, size_t lenbuffer, SCSCP_status* status )
```

It writes the array `buffer` of `lenbuffer` bytes as an OpenMath byte array `<OMB/>...<OMB>` to the `stream`. The argument `buffer` can't be NULL.

This function must only be called by the callback function `callbackwritearg` provided to the hook functions `SCSCP_ss_sendcompletedhook`, `SCSCP_sc_callsendhook`, ....

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value.

If `SCSCP_io_writeOMB` fails, the value of `status` is set to the corresponding error value. `SCSCP_STATUS_IGNORE` could be used for `status` in order to ignore the returned value.

6.5.12 SCSCP_io_writebeginOMA

```c
int SCSCP_io_writebeginOMA ( SCSCP_io* stream, const char* id, SCSCP_status* status )
```

It writes the beginning of the structured Open Math object `<OMA>` to the stream `stream`. `id` is the id of this object for the future reference (see OMR). `id` could be NULL if unset.

This function must only be called by the callback function `callbackwritearg` provided to the hook functions `SCSCP_ss_sendcompletedhook`, `SCSCP_sc_callsendhook`, ....

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If `SCSCP_io_writebeginOMA` fails, the value of `status` is set to the corresponding error value. `SCSCP_STATUS_IGNORE` could be used for `status` in order to ignore the returned value.
6.5.13 SCSCP_io_writeendOMA

int SCSCP_io_writeendOMA ( SCSCP_io* stream, [Library Function] 
SCSCP_status* status )

It writes the end of the structured Open Math object </OMA> to the stream.

This function must only be called by the callback function callbackwritearg provided to
the hook functions SCSCP_ss_sendcompletedhook, SCSCP_sc_callsendhook, ....

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value.
If SCSCP_io_writeendOMA fails, the value of status is set to the corresponding error value.
SCSCP_STATUS_IGNORE could be used for status in order to ignore the returned value.

6.5.14 SCSCP_io_writebeginOMATP

int SCSCP_io_writebeginOMATP ( SCSCP_io* stream, const [Library Function] 
char *id, SCSCP_status* status )

It writes the beginning of the structured Open Math attribute pair <OMATP> to the stream stream. id is the id of this object for the future reference (see OMR). id could be
NULL if unset.

This function must only be called by the callback function callbackwritearg provided to
the hook functions SCSCP_ss_sendcompletedhook, SCSCP_sc_callsendhook, ....

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value.
If SCSCP_io_writebeginOMATP fails, the value of status is set to the corresponding error value.
SCSCP_STATUS_IGNORE could be used for status in order to ignore the returned value.

6.5.15 SCSCP_io_writeendOMATP

int SCSCP_io_writeendOMATP ( SCSCP_io* stream, [Library Function] 
SCSCP_status* status )

It writes the end of the structured Open Math attribute pair </OMATP> to the stream.

This function must only be called by the callback function callbackwritearg provided to
the hook functions SCSCP_ss_sendcompletedhook, SCSCP_sc_callsendhook, ....

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value.
If SCSCP_io_writeendOMATP fails, the value of status is set to the corresponding error value.
SCSCP_STATUS_IGNORE could be used for status in order to ignore the returned value.

6.5.16 SCSCP_io_writebeginOMATTR

int SCSCP_io_writebeginOMATTR ( SCSCP_io* stream, const [Library Function] 
char *id, SCSCP_status* status )

It writes the beginning of the structured Open Math attribution <OMATTR> to the stream stream. id is the id of this object for the future reference (see OMR). id could be
NULL if unset.

This function must only be called by the callback function callbackwritearg provided to
the hook functions SCSCP_ss_sendcompletedhook, SCSCP_sc_callsendhook, ....

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value.
If SCSCP_io_writebeginOMATTR fails, the value of status is set to the corresponding error value.
SCSCP_STATUS_IGNORE could be used for status in order to ignore the returned value.
6.5.17 SCSCP_io_writeendOMATTR

```c
int SCSCP_io_writeendOMATTR ( SCSCP_io* stream, SCSCP_status* status )
```

This function must only be called by the callback function `callbackwritearg` provided to the hook functions `SCSCP_ss_sendcompletedhook`, `SCSCP_sc_callsendhook`, ....

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If `SCSCP_io_writeendOMATTR` fails, the value of `status` is set to the corresponding error value. `SCSCP_STATUS_IGNORE` could be used for `status` in order to ignore the returned value.

6.5.18 SCSCP_io_writebeginOMBIND

```c
int SCSCP_io_writebeginOMBIND ( SCSCP_io* stream, const char *id, SCSCP_status* status )
```

This function must only be called by the callback function `callbackwritearg` provided to the hook functions `SCSCP_ss_sendcompletedhook`, `SCSCP_sc_callsendhook`, ....

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If `SCSCP_io_writebeginOMBIND` fails, the value of `status` is set to the corresponding error value. `SCSCP_STATUS_IGNORE` could be used for `status` in order to ignore the returned value.

6.5.19 SCSCP_io_writeendOMBIND

```c
int SCSCP_io_writeendOMBIND ( SCSCP_io* stream, SCSCP_status* status )
```

This function must only be called by the callback function `callbackwritearg` provided to the hook functions `SCSCP_ss_sendcompletedhook`, `SCSCP_sc_callsendhook`, ....

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If `SCSCP_io_writeendOMBIND` fails, the value of `status` is set to the corresponding error value. `SCSCP_STATUS_IGNORE` could be used for `status` in order to ignore the returned value.

6.5.20 SCSCP_io_writebeginOME

```c
int SCSCP_io_writebeginOME ( SCSCP_io* stream, const char *id, SCSCP_status* status )
```

This function must only be called by the callback function `callbackwritearg` provided to the hook functions `SCSCP_ss_sendcompletedhook`, `SCSCP_sc_callsendhook`, ....

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If `SCSCP_io_writebeginOME` fails, the value of `status` is set to the corresponding error value. `SCSCP_STATUS_IGNORE` could be used for `status` in order to ignore the returned value.
6.5.21 SCSCP_io_writeendOME

```c
int SCSCP_io_writeendOME ( SCSCP_io* stream, SCSCP_status* status )

[Library Function]
```

It writes the end of the structured Open Math error </OME> to the stream.

This function must only be called by the callback function `callbackwritearg` provided to the hook functions `SCSCP_ss_sendcompletedhook`, `SCSCP_sc_callsendhook`, ....

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If `SCSCP_io_writeendOME` fails, the value of `status` is set to the corresponding error value. `SCSCP_STATUS_IGNORE` could be used for `status` in order to ignore the returned value.

6.5.22 SCSCP_io_writebeginOMFOREIGN

```c
int SCSCP_io_writebeginOMFOREIGN ( SCSCP_io* stream, const char *id, SCSCP_status* status )

[Library Function]
```

It writes the beginning of the structured Open Math foreign object </OMFOREIGN> to the stream `stream`. `id` is the id of this object for the future reference (see OMR). `id` could be NULL if unset. Currently, the binary encoding isn’t supported, the function always fails.

This function must only be called by the callback function `callbackwritearg` provided to the hook functions `SCSCP_ss_sendcompletedhook`, `SCSCP_sc_callsendhook`, ....

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If `SCSCP_io_writebeginOMFOREIGN` fails, the value of `status` is set to the corresponding error value. `SCSCP_STATUS_IGNORE` could be used for `status` in order to ignore the returned value.

6.5.23 SCSCP_io_writeendOMFOREIGN

```c
int SCSCP_io_writeendOMFOREIGN ( SCSCP_io* stream, SCSCP_status* status )

[Library Function]
```

It writes the end of the structured Open Math foreign object </OMFOREIGN> to the stream. Currently, the binary encoding isn’t supported, the function always fails.

This function must only be called by the callback function `callbackwritearg` provided to the hook functions `SCSCP_ss_sendcompletedhook`, `SCSCP_sc_callsendhook`, ....

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If `SCSCP_io_writeendOMFOREIGN` fails, the value of `status` is set to the corresponding error value. `SCSCP_STATUS_IGNORE` could be used for `status` in order to ignore the returned value.

6.5.24 SCSCP_io_writebeginOMOBJ

```c
int SCSCP_io_writebeginOMOBJ ( SCSCP_io* stream, SCSCP_status* status )

[Library Function]
```

It writes the beginning of the structured Open Math object </OMOBJ> to the stream.

This function must only be called by the callback function `callbackwritearg` provided to the hook functions `SCSCP_ss_sendcompletedhook`, `SCSCP_sc_callsendhook`, ....

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If `SCSCP_io_writebeginOMOBJ` fails, the value of `status` is set to the corresponding error value. `SCSCP_STATUS_IGNORE` could be used for `status` in order to ignore the returned value.
6.5.25 SCSCP_io_writeendOMOBJ

int SCSCP_io_writeendOMOBJ (SCSCP_io* stream, SCSCP_status* status) [Library Function]

It writes the end of the structured Open Math object </OMOBJ> to the stream.

This function must only be called by the callback function callbackwritearg provided to the hook functions SCSCP_ss_sendcompletedhook, SCSCP_sc_callsendhook, ....

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If SCSCP_io_writeendOMOBJ fails, the value of status is set to the corresponding error value. SCSCP_STATUS_IGNORE could be used for status in order to ignore the returned value.

6.6 Status functions

6.6.1 SCSCP_status_clear

void SCSCP_status_clear (SCSCP_status* status) [Library Function]

It clears the internal structure of the object status and frees allocated memory for this object by any error of the library.

6.6.2 SCSCP_status_is

int SCSCP_status_is (SCSCP_status* status) [Library Function]

This function returns the integer value of the status object. The returned value are defined in See Section 6.2.4 [SCSCP_status], page 10. status mustn't be SCSCP_STATUS_IGNORE. This function could be defined as a macro in the header file scscp.h.

```c
int res = SCSCP_co_get_maxmemory(&options, &memsize, status);
if (res)
{
    printf("maximum memory = %lld\n", (long long)memsize);
}
else if (SCSCP_status_is(status)==SCSCP_STATUS_USAGEUNKNOWNMAXMEMORY)
{
    printf("maximum memory not available\n");
}
```

6.6.3 SCSCP_status_strerror

const char* SCSCP_status_strerror (const SCSCP_status* status) [Library Function]

This function accepts an argument status and returns a pointer to the corresponding message string.
```c
int res = SCSCP_co_get_maxmemory(&options, &memsize, status);
if (res)
{
    printf("maximum memory = %lld\n", (long long)memsize);
}
else
{
    printf("error message =\'%s\n", SCSCP_status_strerror(status));
}
```

6.7 Procedure call options functions

6.7.1 SCSCP_co_init

```c
int SCSCP_co_init (SCSCP_calloptions* options, SCSCP_status* status) {
    It initializes the internal structure of the object options. It generates and sets the call identifier of the options of the procedure call. This call identifier is defined as the symbol call_id of the OpenMath Content Dictionary scscp1. The call identifier is prefixed by libSCSCP:. It sets the return type to the value SCSCP_option_return_object.

    On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If SCSCP_co_init fails, the value of status is set to the corresponding error value. SCSCP_STATUS_IGNORE could be used for status in order to ignore the returned value.
```

6.7.2 SCSCP_co_clear

```c
int SCSCP_co_clear (SCSCP_calloptions* options, SCSCP_status* status) {
    It clears the internal structure of the object options and frees allocated memory for this object by the function SCSCP_co_init.

    On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If SCSCP_co_clear fails, the value of status is set to the corresponding error value. SCSCP_STATUS_IGNORE could be used for status in order to ignore the returned value.
```

6.7.3 SCSCP_co_set_callid

```c
int SCSCP_co_set_callid (SCSCP_calloptions* options, const char *buffer, SCSCP_status* status) {
    This function sets the call identifier of the options of the procedure call. It overwrites the default call identifier, generated by SCSCP_co_init. This call identifier is defined as the symbol call_id of the OpenMath Content Dictionary scscp1.

    The argument buffer can’t be NULL and won’t be duplicated by these functions. So buffer mustn’t be destroyed until the function SCSCP_co_clear is called on the object options. The argument buffer shouldn’t be use any string beginning with libSCSCP: because SCSCP_co_init generates unique call identifier with this prefix. The caller of this function is responsible of the unicity of the content of buffer during th connection.
```
On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If SCSCP_co_set_callid fails, the value of status is set to the corresponding error value. SCSCP_STATUS_IGNORE could be used for status in order to ignore the returned value.

6.7.4 SCSCP_co_get_callid

```c
int SCSCP_co_get_callid ( SCSCP_calloptions* options, const char **buffer, SCSCP_status* status )
```

This function returns, in the argument buffer, the call id of the options of the procedure call. This call id is defined as the symbol call_id of the OpenMath Content Dictionary scscp1. The argument buffer can’t be NULL. The returned string mustn’t be modified.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If SCSCP_co_get_callid fails, the value of status is set to the corresponding error value. SCSCP_STATUS_IGNORE could be used for status in order to ignore the returned value.

6.7.5 SCSCP_co_set_runtimelimit

```c
int SCSCP_co_set_runtimelimit ( SCSCP_calloptions* options, size_t time, SCSCP_status* status )
```

This function sets the amount of time in milliseconds, with the value time, that the SCSCP server should spend on this call. This runtime limit is defined by the symbol option_runtime of the OpenMath Content Dictionary scscp1.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If SCSCP_co_set_runtimelimit fails, the value of status is set to the corresponding error value. SCSCP_STATUS_IGNORE could be used for status in order to ignore the returned value.

6.7.6 SCSCP_co_get_runtimelimit

```c
int SCSCP_co_get_runtimelimit ( SCSCP_calloptions* options, size_t* time, SCSCP_status* status )
```

This function returns, in the argument time, the amount of time in milliseconds that the server should spend on this call. This amount of time is defined as the symbol option_runtime of the OpenMath Content Dictionary scscp1. If the amount of time isn’t available (not supplied by the server), the function fails and status is set to SCSCP_STATUS.UsageUnknownRuntimeLimit.

The argument time can’t be NULL.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If SCSCP_co_get_runtimelimit fails, the value of status is set to the corresponding error value. SCSCP_STATUS_IGNORE could be used for status in order to ignore the returned value.

6.7.7 SCSCP_co_set_minmemory

```c
int SCSCP_co_set_minmemory ( SCSCP_calloptions* options, size_t memsize, SCSCP_status* status )
```

This function sets the minimum amount of memory in bytes, with the value memsize, that the server should use on this call. This memory limit is defined by the symbol option_min_memory of the OpenMath Content Dictionary scscp1.
On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If SCSCP_co_set_minmemory fails, the value of status is set to the corresponding error value. SCSCP_STATUS_IGNORE could be used for status in order to ignore the returned value.

### 6.7.8 SCSCP_co_get_minmemory

```c
int SCSCP_co_get_minmemory ( SCSCP_calloptions* options, [Library Function]
    size_t* memsize, SCSCP_status* status )
```

This function returns, in the argument memsize, the minimum amount of memory in bytes that the server should use on this call. This amount of memory is defined as the symbol option_min_memory of the OpenMath Content Dictionary scscp1. If the amount of memory isn’t available (not supplied by the server), the function fails and status is set to SCSCP_STATUS_USAGEUNKNOWNMINMEMORY.

The argument memsize can’t be NULL.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If SCSCP_co_get_minmemory fails, the value of status is set to the corresponding error value. SCSCP_STATUS_IGNORE could be used for status in order to ignore the returned value.

### 6.7.9 SCSCP_co_set_maxmemory

```c
int SCSCP_co_set_maxmemory ( SCSCP_calloptions* options, [Library Function]
    size_t memsize, SCSCP_status* status )
```

This function sets the maximum amount of memory in bytes, with the value memsize, that the SCSCP server should use on this call. This memory limit is defined by the symbol option_max_memory of the OpenMath Content Dictionary scscp1.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If SCSCP_co_set_maxmemory fails, the value of status is set to the corresponding error value. SCSCP_STATUS_IGNORE could be used for status in order to ignore the returned value.

### 6.7.10 SCSCP_co_get_maxmemory

```c
int SCSCP_co_get_maxmemory ( SCSCP_calloptions* options, [Library Function]
    size_t* memsize, SCSCP_status* status )
```

This function returns, in the argument memsize, the maximum amount of memory in bytes that the server should use on this call. This amount of memory is defined as the symbol option_max_memory of the OpenMath Content Dictionary scscp1. If the amount of memory isn’t available (not supplied by the server), the function fails and status is set to SCSCP_STATUS_USAGEUNKNOWNMAXMEMORY.

The argument memsize can’t be NULL.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If SCSCP_co_get_maxmemory fails, the value of status is set to the corresponding error value. SCSCP_STATUS_IGNORE could be used for status in order to ignore the returned value.
6.7.11 SCSCP_co_set_debuglevel

int SCSCP_co_set_debuglevel ( SCSCP_calloptions* options, int debuglevel, SCSCP_status* status )

This function sets the debug level, with the value debuglevel, that the client is interested. This debug level is defined by the symbol `option_debuglevel` of the OpenMath Content Dictionary `scscp1`.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If `SCSCP_co_set_debuglevel` fails, the value of `status` is set to the corresponding error value. `SCSCP_STATUS_IGNORE` could be used for `status` in order to ignore the returned value.

6.7.12 SCSCP_co_get_debuglevel

int SCSCP_co_get_debuglevel ( SCSCP_calloptions* options, int* debuglevel, SCSCP_status* status )

This function returns, in the argument `debuglevel`, the debug level that the client is interested. This debug level is defined as the symbol `option_max_memory` of the OpenMath Content Dictionary `scscp1`. If the debug level isn’t available (not supplied by the server), the function fails and `status` is set to `SCSCP_STATUS_USAGEUNKNOWNDEBUGLEVEL`.

The argument `debuglevel` can’t be NULL.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If `SCSCP_co_get_debuglevel` fails, the value of `status` is set to the corresponding error value. `SCSCP_STATUS_IGNORE` could be used for `status` in order to ignore the returned value.

6.7.13 SCSCP_co_set_returntype

int SCSCP_co_set_returntype ( SCSCP_calloptions* options, SCSCP_option_return returntype, SCSCP_status* status )

This function sets the return type, with the value `returntype`, of the procedure call that the server should send. The available value for `returntype` are

- `SCSCP_option_return_object`. The return value is an OpenMath object. It’s defined by the symbol `option_return_object` of the OpenMath Content Dictionary `scscp1`.
- `SCSCP_option_return_cookie`. The return value is a cookie (a reference to an OpenMath object). It’s defined by the symbol `option_return_cookie` of the OpenMath Content Dictionary `scscp1`.
- `SCSCP_option_return_nothing`. The procedure call returns no value. It’s defined by the symbol `option_return_nothing` of the OpenMath Content Dictionary `scscp1`.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If `SCSCP_co_set_returntype` fails, the value of `status` is set to the corresponding error value. `SCSCP_STATUS_IGNORE` could be used for `status` in order to ignore the returned value.

6.7.14 SCSCP_co_get_returntype

int SCSCP_co_get_returntype ( SCSCP_calloptions* options, SCSCP_option_return* returntype, SCSCP_status* status )

This function returns, in the argument `returntype`, the return type of the "procedure call" message that the server should send. The possible value of `returntype` are described in
the function `SCSCP_co_set_returntype`. If the return type isn’t available (not supplied by the server), the function fails and status is set to `SCSCP_STATUS_USAGEUNKNOWNRETURNTYPE`.

The argument `returntype` can’t be NULL.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If `SCSCP_co_get_returntype` fails, the value of status is set to the corresponding error value. `SCSCP_STATUS_IGNORE` could be used for status in order to ignore the returned value.

### 6.7.15 SCSCP_co_get_encodingtype

```c
int SCSCP_co_get_encodingtype ( SCSCP_calloptions* options, SCSCP_encodingtype* encodingtype, SCSCP_status* status )
```

This function returns, in the argument `encodingtype`, the current encoding for the OpenMath objects associated with this "procedure call" message.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If `SCSCP_co_get_encodingtype` fails, the value of `status` is set to the corresponding error value. `SCSCP_STATUS_IGNORE` could be used for `status` in order to ignore the returned value.

### 6.8 Procedure return options functions

#### 6.8.1 SCSCP_ro_init

```c
int SCSCP_ro_init ( SCSCP_returnoptions* options, SCSCP_status* status )
```

It initializes the internal structure of the object `options`.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If `SCSCP_ro_init` fails, the value of `status` is set to the corresponding error value. `SCSCP_STATUS_IGNORE` could be used for `status` in order to ignore the returned value.

#### 6.8.2 SCSCP_ro_clear

```c
int SCSCP_ro_clear ( SCSCP_returnoptions* options, SCSCP_status* status )
```

It clears the internal structure of the object `options` and frees allocated memory for this object by the function `SCSCP_ro_init`.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If `SCSCP_ro_clear` fails, the value of `status` is set to the corresponding error value. `SCSCP_STATUS_IGNORE` could be used for `status` in order to ignore the returned value.

#### 6.8.3 SCSCP_ro_set_callid

```c
int SCSCP_ro_set_callid ( SCSCP_returnoptions* options, const char *buffer, SCSCP_status* status )
```

This function sets the call id, with the value `buffer`, of the options of the procedure return. This call id is defined as the symbol `call_id` of the OpenMath Content Dictionary `scscp1`. The argument `buffer` can’t be NULL and won’t be duplicated by this function. So `buffer` can’t be destroyed until the function `SCSCP_ro_clear` is called on the object `options`. 
On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If SCSCP_ro_set_callid fails, the value of status is set to the corresponding error value. SCSCP_STATUS_IGNORE could be used for status in order to ignore the returned value.

### 6.8.4 SCSCP_ro_get_callid

```c
int SCSCP_ro_get_callid ( SCSCP_returnoptions* options, const char **buffer, SCSCP_status* status )
```

This function returns, in the argument buffer, the call id of the options of the procedure return. This call id is defined as the symbol call_id of the OpenMath Content Dictionary scscp1. The argument buffer mustn’t be NULL. The returned string mustn’t be modified.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If SCSCP_ro_get_callid fails, the value of status is set to the corresponding error value. SCSCP_STATUS_IGNORE could be used for status in order to ignore the returned value.

### 6.8.5 SCSCP_ro_set_runtime

```c
int SCSCP_ro_set_runtime ( SCSCP_returnoptions* options, size_t time, SCSCP_status* status )
```

This function sets the amount of time in milliseconds, with the value time, that the server spent on this call. This amount of time is defined as the symbol info_runtime of the OpenMath Content Dictionary scscp1.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If SCSCP_ro_set_runtime fails, the value of status is set to the corresponding error value. SCSCP_STATUS_IGNORE could be used for status in order to ignore the returned value.

### 6.8.6 SCSCP_ro_get_runtime

```c
int SCSCP_ro_get_runtime ( SCSCP_returnoptions* options, size_t* time, SCSCP_status* status )
```

This function returns, in the argument time, the amount of time in milliseconds that the server spent on this call. This amount of time is defined as the symbol info_runtime of the OpenMath Content Dictionary scscp1. If the amount of time isn’t available (not supplied by the server), the function fails and status is set to SCSCP_STATUS_USAGEUNKNOWNRUNTIME.

The argument time can’t be NULL.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If SCSCP_ro_get_runtime fails, the value of status is set to the corresponding error value. SCSCP_STATUS_IGNORE could be used for status in order to ignore the returned value.

### 6.8.7 SCSCP_ro_set_memory

```c
int SCSCP_ro_set_memory ( SCSCP_returnoptions* options, size_t memsize, SCSCP_status* status )
```

This function sets the amount of memory in bytes, with the value memsize, that the server used on this call. This amount of memory is defined as the symbol info_memory of the OpenMath Content Dictionary scscp1.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If SCSCP_ro_set_memory fails, the value of status is set to the corresponding error value. SCSCP_STATUS_IGNORE could be used for status in order to ignore the returned value.
6.8.8 SCSCP_ro_get_memory

```c
int SCSCP_ro_get_memory ( SCSCP_returnoptions* options, [Library Function]
    size_t* memsize, SCSCP_status* status )
```

This function returns, in the argument `memsize`, the amount of memory in bytes that
the server used for this call. This amount of memory is defined as the symbol `info_ 
memory` of the OpenMath Content Dictionary `scscp1`. If the amount of memory isn’t
available (not supplied by the server), the function fails and `status` is set to `SCSCP_STATUS_ 
USAGEUNKNOWNMEM`.

The argument `memsize` can’t be NULL.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value.
If `SCSCP_ro_get_runtime` fails, the value of `status` is set to the corresponding error value.
`SCSCP_STATUS_IGNORE` could be used for `status` in order to ignore the returned value.

6.8.9 SCSCP_ro_set_message

```c
int SCSCP_ro_set_message ( SCSCP_returnoptions* options, [Library Function]
    const char *buffer, SCSCP_status* status )
```

This function sets the information message, with the value `buffer`, of the options of the
procedure return. This information message is defined as the symbol `info_message` of the
OpenMath Content Dictionary `scscp1`. The argument `buffer` can’t be NULL and won’t be
duplicated by this function. So `buffer` can’t be destroyed until the function `SCSCP_ro_clear`
is called on the object `options`.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value.
If `SCSCP_ro_set_message` fails, the value of `status` is set to the corresponding error value.
`SCSCP_STATUS_IGNORE` could be used for `status` in order to ignore the returned value.

6.8.10 SCSCP_ro_get_message

```c
int SCSCP_ro_get_message ( SCSCP_returnoptions* options, [Library Function]
    const char **buffer, SCSCP_status* status )
```

This function returns, in the argument `buffer`, the information message of the options of
the procedure return. This information message is defined as the symbol `info_message` of
the OpenMath Content Dictionary `scscp1`. The argument `buffer` mustn’t be NULL. The
returned string mustn’t be modified.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value.
If `SCSCP_ro_get_message` fails, the value of `status` is set to the corresponding error value.
`SCSCP_STATUS_IGNORE` could be used for `status` in order to ignore the returned value.

6.9 XML parsing functions

The following functions are useful to parse incoming messages after reading the header of
the message with the functions `SCSCP_sc_callrecvheader` or `SCSCP_ss_callrecvheader`.
The following example prints each node and attribute of a XML tree.
void printelements(SCSCP_xmlnodeptr node, int tab)  
{
    SCSCP_xmlattrptr attr;
    const char *name;
    const char *value;
    int j;

    while (node!=NULL)
    {
        for(j=0; j<tab; j++) putchar(' ');
        printf("node : '%s'
",SCSCP_xmlnode_getname(node));

        for (attr = SCSCP_xmlnode_getattr(node);
             attr!=NULL;
             attr = SCSCP_xmlattr_getnext(attr))
        {
            SCSCP_xmlattr_getvalue(attr, &name, &value);
            for(j=0; j<tab+1; j++) putchar(' ');
            printf("attribute : '%s' = '%s'
",name, value);
        }
        printelements(SCSCP_xmlnode_getchild(node),tab+4);
        node = SCSCP_xmlnode_getnext(node);
    }
}

6.9.1 SCSCP_xmlnode_getnext

SCSCP_xmlnodeptr SCSCP_xmlnode_getnext ( [Library Function]  
                              SCSCP_xmlnodeptr curnode )

This function returns a pointer to the next node. If curnode is the last element, this
function returns NULL.

6.9.2 SCSCP_xmlnode_getname

const char* SCSCP_xmlnode_getname ( [Library Function]  
                                      SCSCP_xmlnodeptr curnode )

This function returns the name of the node curnode.

6.9.3 SCSCP_xmlnode_getid

const char* SCSCP_xmlnode_getid ( [Library Function]  
                                     SCSCP_xmlnodeptr curnode )

This function returns the reference id (<OM... id="...">) of the node curnode. The
function returns NULL if the reference id isn’t available.
6.9.4 SCSCP_xmlnode_getchild

SCSCP_xmlnodeptr SCSCP_xmlnode_getchild ( )
[Library Function]

This function returns the first child of the node curnode. The function returns NULL if it has no child.

6.9.5 SCSCP_xmlnode_getcontent

const char * SCSCP_xmlnode_getcontent ( )
[Library Function]

This function returns as a string the content of the node curnode.

6.9.6 SCSCP_xmlnode_getattr

SCSCP_xmlattrptr SCSCP_xmlnode_getattr ( )
[Library Function]

This function returns the first attribute of the node curnode.

6.9.7 SCSCP_xmlattr_getnext

SCSCP_xmlattrptr SCSCP_xmlattr_getnext ( )
[Library Function]

This function returns a pointer to the next attribute. If attr is the last attribute, this function returns NULL.

6.9.8 SCSCP_xmlattr_getvalue

int SCSCP_xmlattr_getvalue ( )
[Library Function]

This function returns, in the argument name, a pointer to the name of the attribute attr and returns, in the argument value, a pointer to the value of the attribute attr.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value.

6.9.9 SCSCP_xmlnode_readOMS

int SCSCP_xmlnode_readOMS ( )
[Library Function]

This function reads an OpenMath symbol and the cd name to cdname and the name to symbolname. symbolname and cdname mustn’t be NULL.

On exit, it returns 0 if node doesn’t contain an OpenMath symbol, otherwise the return value is a non-zero value.

6.9.10 SCSCP_xmlnode_readOMIstr

int SCSCP_xmlnode_readOMIstr ( )
[Library Function]

This function reads an OpenMath integer and stores it to value. value mustn’t be NULL.

On exit, it returns 0 if node doesn’t contain an OpenMath integer, otherwise the return value is a non-zero value.
6.9.11 SCSCP_xmlnode_readOMIInt

```c
int SCSCP_xmlnode_readOMIInt ( SCSCP_xmlnodeptr* node,  
  int* value)  [Library Function]
```

This function reads an OpenMath integer and store it to `value`.

On exit, it returns 0 if `node` doesn’t contain an OpenMath integer, otherwise the return value is a non-zero value.

6.9.12 SCSCP_xmlnode_readOMFstr

```c
int SCSCP_xmlnode_readOMFstr ( SCSCP_xmlnodeptr* node,  
  const char ** value, int * base)  [Library Function]
```

This function reads an OpenMath floating-point number and stores it to `value`. It stores to `base` the base used to represent the floating-point number (16 for "hex", 10 for "dec"). `value` and `base` mustn’t be NULL.

On exit, it returns 0 if `node` doesn’t contain an OpenMath floating-point, otherwise the return value is a non-zero value.

6.9.13 SCSCP_xmlnode_readOMFdouble

```c
int SCSCP_xmlnode_readOMFdouble ( SCSCP_xmlnodeptr* node,  
  double* value)  [Library Function]
```

This function reads an OpenMath floating-point number and stores it to `value`. `value` mustn’t be NULL.

On exit, it returns 0 if `node` doesn’t contain an OpenMath floating-point, otherwise the return value is a non-zero value.

6.9.14 SCSCP_xmlnode_readOMSTR

```c
int SCSCP_xmlnode_readOMSTR ( SCSCP_xmlnodeptr* node,  
  const char ** value)  [Library Function]
```

This function reads an OpenMath string and store it to `value`. `value` mustn’t be NULL.

On exit, it returns 0 if `node` doesn’t contain an OpenMath string, otherwise the return value is a non-zero value.

6.9.15 SCSCP_xmlnode_readOMR

```c
int SCSCP_xmlnode_readOMR ( SCSCP_xmlnodeptr* node,  
  char ** value)  [Library Function]
```

This function reads an OpenMath reference and stores it to `value`. `value` mustn’t be NULL.

On exit, it returns 0 if `node` doesn’t contain an OpenMath reference, otherwise the return value is a non-zero value.

6.9.16 SCSCP_xmlnode_readOMV

```c
int SCSCP_xmlnode_readOMV ( SCSCP_xmlnodeptr* node,  
  char ** value)  [Library Function]
```

This function reads an OpenMath variable and stores it to `value`. `value` mustn’t be NULL.
On exit, it returns 0 if node doesn’t contain an OpenMath variable, otherwise the return value is a non-zero value.

6.9.17 SCSCP_xmlnode_readpairOMSOMSTR

```c
int SCSCP_xmlnode_readpairOMSOMSTR (SCSCP_xmlnodeptr* node, const char *cdname, const char *symbolname, const char **buffer)
```

This function reads an OpenMath symbol and an OpenMath string. It checks that the symbol has the same symbol name and cd name as the arguments symbolname and cdname. It stores the OpenMath string to value. cdname, symbolname and value mustn’t be NULL.

On exit, it returns 0 if node doesn’t contain OpenMath symbol and an OpenMath string, otherwise the return value is a non-zero value.

6.10 Remote objects functions

6.10.1 SCSCP_sc_remoteobjectstoresessionhook

```c
int SCSCP_sc_remoteobjectstoresessionhook (SCSCP_socketclient* client, int (*callbackwriteargs)(SCSCP_io* stream, void *param, SCSCP_status* status), void* param, char **cookiename, SCSCP_status* status)
```

The client sends a "procedure call" message to store an object on the server. This object will be usable in the remainder of the current SCSCP session. The object is written by the callback function callbackwriteargs. The function callbackwriteargs must use the functions SCSCP_io_writexxx, such as SCSCP_io_writeOMSTR, to write data which are sent to the SCSCP server. A connection must be previously opened with SCSCP_sc_connect before performing this call.

The argument param is a pointer which is provided to the callbackwriteargs to exchange information. This pointer and its content isn’t modified by SCSCP_sc_remoteobjectstoresessionhook.

On exit, the argument cookiename contains the name of the Openmath reference returned by the server. This string must be freed by the system call free.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If SCSCP_sc_remoteobjectstoresessionhook fails, the value of status is set to the corresponding error value. SCSCP_STATUS_IGNORE could be used for status in order to ignore the returned value.

6.10.2 SCSCP_sc_remoteobjectstorepersistenthook

```c
int SCSCP_sc_remoteobjectstorepersistenthook (SCSCP_socketclient* client, int (*callbackwriteargs)(SCSCP_io* stream, void *param, SCSCP_status* status), void* param, char **cookiename, SCSCP_status* status)
```

The client sends a "procedure call" message to store an object on the server. This object will be usable from different SCSCP session. The object is written by the callback
function \textit{callbackwriteargs}. The function \textit{callbackwriteargs} must use the functions \texttt{SCSCP\_io\_writexxx}, such as \texttt{SCSCP\_io\_writeOMSTR}, to write data which are sent to the SCSCP server. A connection must be previously opened with \texttt{SCSCP\_sc\_connect} before preforming this call.

The argument \textit{param} is a pointer which is provided to the \textit{callbackwriteargs} to exchange information. This pointer and its content isn’t modified by \texttt{SCSCP\_sc\_remoteobjectstorepersistenthook}.

On exit, the argument \textit{cookiename} contains the name of the Openmath reference returned by the server. This string must be freed by the system call \texttt{free}.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If \texttt{SCSCP\_sc\_remoteobjectstorepersistenthook} fails, the value of \textit{status} is set to the corresponding error value. \texttt{SCSCP\_STATUS\_IGNORE} could be used for \textit{status} in order to ignore the returned value.

\textbf{6.10.3 SCSCP\_sc\_remoteobjectretrievexmlnode}

\begin{verbatim}
int SCSCP_sc_remoteobjectretrievexmlnode ( SCSCP_socketclient* client, const char *cookiename, SCSCP_xmlnodeptr* node, SCSCP_status* status)

The \textit{client} sends a "procedure call" message to retrieve the value of an remote object \texttt{cookiename} from the server. A connection must be previously opened with \texttt{SCSCP\_sc\_connect} before preforming this call. This function returns, in the argument \textit{node}, a pointer to this OpenMath object.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If \texttt{SCSCP\_sc\_remoteobjectretrievexmlnode} fails, the value of \textit{status} is set to the corresponding error value. \texttt{SCSCP\_STATUS\_IGNORE} could be used for \textit{status} in order to ignore the returned value.

\textbf{6.10.4 SCSCP\_sc\_remoteobjectretrievestr}

\begin{verbatim}
int SCSCP_sc_remoteobjectretrievestr ( SCSCP_socketclient* client, const char *cookiename, char** openmathbuffer, SCSCP_status* status)

The \textit{client} sends a "procedure call" message to retrieve the value of an remote object \texttt{cookiename} from the server. A connection must be previously opened with \texttt{SCSCP\_sc\_connect} before preforming this call.

This function returns, in the argument \texttt{openmathbuffer}, a string of this OpenMath object. This string must be freed by the system call \texttt{free}.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If \texttt{SCSCP\_sc\_remoteobjectretrievestr} fails, the value of \textit{status} is set to the corresponding error value. \texttt{SCSCP\_STATUS\_IGNORE} could be used for \textit{status} in order to ignore the returned value.
\end{verbatim}
6.10.5 SCSCP_sc_remoteobjectunbind

```c
int SCSCP_sc_remoteobjectunbind ( SCSCP_socketclient* client, const char *cookiename, SCSCP_status* status)
```

The `client` sends a "procedure call" message to remove the remote object `cookiename` from the server. A connection must be previously opened with `SCSCP_sc_connect` before performing this call.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value. If `SCSCP_sc_remoteobjectunbind` fails, the value of `status` is set to the corresponding error value. `SCSCP_STATUS_IGNORE` could be used for `status` in order to ignore the returned value.

6.11 Miscellaneous functions

6.11.1 SCSCP_get_libversionstr

```c
const char* SCSCP_get_libversionstr (void)
```

Return the SCSCP C Library version, as a null-terminated string.

6.11.2 SCSCP_get_libversionnum

```c
void SCSCP_get_libversionnum (unsigned int *major, unsigned int *minor, unsigned int *patch)
```

Return the SCSCP C Library version as unsigned values. On exit, `major` contains the major revision, `minor` contains the minor revision and `patch` the patch level revision.

6.11.3 SCSCP_CHECK_LIBVERSION

```c
int SCSCP_CHECK_LIBVERSION ()
```

Verify that the SCSCP C Library versions are consistent.

It verifies that the version of the SCSCP C Library with which an application was compiled matches the version of the SCSCP C Library against which the application is currently linked.

This function is defined as a macro and returns an integer of type int.

On exit, it returns

- 0 if the version of the library are not compatible,
- 1 if the version of the library are compatible and are exactly the same.
- 2 if the version of the library are compatible and are not exactly the same.
7 Design a SCSCP C server

The file `examples/decodeserver.c` shows the server which decodes each node of the OpenMath expression received from the client. It sends an answer to the client depending on the call options.

A simple SCSCP server could be done with the following operations. This server supports the scscp versions "1.3" and "1.5beta".

- **Initialize the server**

```c
SCSCP_socketserver server;
SCSCP_status status = SCSCP_STATUS_INITIALIZER;
SCSCP_ss_init(&server, &status, "MYCAS", "1.0", "myid",
            SCSCP_PROTOCOL_VERSION_1_3,
            "1.5beta",
            NULL);
```

- **Listen for incoming client**

```c
SCSCP_ss_listen(&client, SCSCP_PROTOCOL_DEFAULTPORT, 0, &status);
```

- **Loop over new clients**

```c
SCSCP_incomingclient incomingclient;
while (SCSCP_ss_acceptclient(&server, &incomingclient, &status))
{
    ...
```

- **Receive the "procedure call" message : 2 solutions**

```c
SCSCP_calloptions calloptions;
SCSCP_returnoptions returnoptions;
SCSCP_msgtype msgtype;

SCSCP_co_init(&calloptions, status);
SCSCP_ro_init(&returnopt, status);
```

- **solution 1 : read the header and decode each node of the openmath stream**

```c
SCSCP_ss_callrecvheader(&incomingclient, &calloptions,
          &msgtype, &status);
```

```c
SCSCP_xmlnodeptr ptrnode;
ptrnode = SCSCP_ss_getxmlnode(&incomingclient, &status);
```

- **solution 2 : read the header and store the content in a string buffer**

```c
char *openmathbuffer;
SCSCP_ss_callrecvstr(&incomingclient, &calloptions, &msgtype,
            &openmathbuffer, &status);
```

- **Send the answer : procedure completed or terminated ?**

```c
    const char *openmathanswer="<OM...>";
```
SCSCP_ss_sendcompletedstr(&incomingclient, &returnopt, openmathanswer, &status);

- Send a "procedure terminated" message
  
  const char *messageerror="can’t store an object";
  
  SCSCP_ss_sendterminatedstr(&incomingclient, &returnopt, "sccsp1", "error_system_specific", messageerror, &status);

- clear the options object
  SCSCP_co_clear(&options, status);
  SCSCP_ro_clear(&returnopt, status);

- Close the connection
  SCSCP_ss_closeincoming(&incomingclient, &status);

- Stop to listen for incoming clients
  SCSCP_ss_close(&server, &status);

- Clear the server
  SCSCP_ss_clear(&server, &status);
8 Design a SCSCP C client

The file `examples/simplestclient.c` shows the simplest client which stores the value 6177887 on the server and prints the answer of the server.

The file `examples/decodeclient.c` shows the client which decodes each node of the OpenMath expression received from the server.

A simple SCSCP client could be done with the following operations. This simple client will connect with the SCSCP server located on "localhost" and listening on port 26133.

- Initialize the client
  ```c
  SCSCP_socketclient client;
  SCSCP_status status = SCSCP_STATUS_INITIALIZER;
  SCSCP_sc_init(&client, &status, SCSCP_PROTOCOL_VERSION_1_3, NULL);
  ```

- Open the Connection
  ```c
  SCSCP_sc_connect(&client, "localhost",
                  SCSCP_PROTOCOL_DEFAULTPORT, &status);
  ```

- Send a procedure call
  ```c
  SCSCP_calloptions calloptions;
  SCSCP_co_init(&calloptions, status);
  SCSCP_sc_callsendstr(&client, &calloptions, 
                      "<OMA><OMS cd="scscp2" name="get_allowed_heads" /></OMA>",
                      &status);
  ```

- Receive the answer of the procedure call : 2 solutions
  - solution 1 : read the header and decode each node of the openmath stream
    ```c
    SCSCP_msgtype msgtype;
    SCSCP_returnoptions options;
    SCSCP_ro_init(&options, status);
    SCSCP_sc_callrecvheader(client, &options, &msgtype, &status);
    if (msgtype==SCSCP_msgtype_ProcedureTerminated)
    {
        char *messagebuffer;
        char *cdname;
        char *symbolname;
        SCSCP_sc_callrecvterminated(client, &cdname, &symbolname,
                                    &messagebuffer, status);
    }
    ```
  - solution 2 : read the header and store the content in a string buffer
    ```c
    SCSCP_msgtype msgtype;
    SCSCP_returnoptions options;
    char *buffer;
    SCSCP_ro_init(&options, status);
    ```
SCSCP_sc_callrecvstr(client, &options, &msgtype,
                     &buffer, &status);

- Close the connection
  SCSCP_sc_close(&client,&status);
- Clear the client
  SCSCP_sc_clear(&client,&status);
9 C++ Interface

All C++ functions and class are grouped in the namespace SCSCP.

9.1 Server

9.1.1 Constructor

Server (const char* servicename, const char* serviceversion, const char* serviceid) throw(Exception)

It initializes the SCSCP server with the default versions (SCSCP_PROTOCOL_VERSION_1_3 and SCSCP_PROTOCOL_VERSION_1_2) of SCSCP protocol that could be negotiated. The arguments servicename, serviceversion and serviceid are used as the value of the attribute service_name, service_version and service_id of the Connection Initiation Message.

~Server (void) [Destructor]
If the server was listening on a port, the method Server::close is called before clearing the object. It destroys the SCSCP server.

9.1.2 listen

int listen (int port=SCSCP_PROTOCOL_DEFAULTPORT) throw(Exception)

It creates an internal queue for the incoming connections and starts to listen on the port of "localhost". If the port isn’t available, it retries with the next port until it finds an available port. If port is 0, an available random port is chosen.

On exit, if an error occurs, then it returns 0 and an exception is raised if the exception mechanism is enabled. Otherwise the return value is a non-zero value.

9.1.3 getport

int getport () const [Method on Server]
It returns the port from the server is listening. If the function listen failed or is not called before, then this function returns -1.

9.1.4 close

int close () throw(Exception) [Method on Server]
It terminates to listen for the incoming connections.

On exit, if an error occurs, then it returns 0 and an exception is raised if the exception mechanism is enabled. Otherwise the return value is a non-zero value.

9.1.5 eof

bool eof () const [Method on Server]
It returns true if the connection is closed.
9.1.6 acceptclient

**IncomingClient* acceptclient () throw ()** [Method on Server]

It extracts the first connection request on the queue of pending connections. If no pending connections are present on the queue, it blocks the caller until a connection is present.

After the *Connection Initiation*, the server returns an object to manage future exchanged messages.

After the transactions, the returned pointer must be deleted to close the connection and release the memory.

On exit, if an error occurs, then it returns 0 and an exception is raised if the exception mechanism is enabled. Otherwise the return value is a non-zero value.

The following example shows how to implement the main loop of the SCSCP C++ server.

```
SCSCP::Server server("MYCAS","1","myid");
SCSCP::IncomingClient *incomingclient;

/*listen on the default port */
server.listen();

while ((incomingclient=server.acceptclient())!=NULL)
{
    ... process incoming messages ...
    delete incomingclient;
}
/* stop the server */
server.close();
```

9.2 IncomingClient

The class `IncomingClient` handles a connection on the SCSCP server side. So on the server side, several instances of that class could exist at the same time to handle connection from different clients. That class is only instanciated by the function `Server::acceptclient()`.

On the server side, the "Procedure Call" request are handled by an instance of `Server::Computation`.

9.2.1 Destructor

```IncomingClient (void) [Destructor]
If a connection with a client was already opened, the connection is closed with the client.```
9.2.2 eof

bool eof () const

It returns true if the connection is closed.

9.3 Client

The class Client handles the connection on the SCSCP client side. After a connection is initialized, the procedure call could be performed using an instance of the class Client::Computation (see Section 9.4 [Procedure call], page 52).

9.3.1 Constructor

Client (void) throw(Exception)

It initializes the SCSCP client with the default version (SCSCP_PROTOCOL_VERSION_1.3 and SCSCP_PROTOCOL_VERSION_1.2) of SCSCP protocol that could be negotiated.

During the negotiation with the server, the client will choose the first version, that the server supports too, in the following order: SCSCP_PROTOCOL_VERSION_1.3, and SCSCP_PROTOCOL_VERSION_1.2.

~Client (void)

If a connection was already opened, the method Client::close is called before clearing the object. It destroys the SCSCP client.

9.3.2 connect

int connect (const char *machine, int port=SCSCP_PROTOCOL_DEFAULTPORT) throw(Exception)

It tries to connect to the SCSCP server which is running on the computer machine and is listening on the port port.

machine could be any string but it must resolved as an IP address. Its value could be "localhost" if the SCSCP server runs on the same computer.

In most of the case, the default value SCSCP_PROTOCOL_DEFAULTPORT should be used for the port number.

On exit, if an error occurs, then it returns 0 and an exception is raised if the exception mechanism is enabled. Otherwise the return value is a non-zero value.

9.3.3 close

int close () throw(Exception)

It closes a connection previously opened by the client with the method Client::connect. On exit, if an error occurs, then it returns 0 and an exception is raised if the exception mechanism is enabled. Otherwise the return value is a non-zero value.

9.3.4 eof

bool eof () const

It returns true if the connection is closed.
9.4 Procedure call

The procedure calls are managed on the client side by a instance of the class Client::Computation. The procedure calls are managed on the server side by a instance of the class Server::Computation. The class Client::Computation and Server::Computation derivate from ProcedureCall and inherit of all public methods of that class.

9.4.1 ProcedureCall

This class should not be instantiated by the application. Only the Client::Computation and Server::Computation should be instantiated. This class provides methods to access the call options or return options of a procedure call.

9.4.1.1 ProcedureCall Constructor

ProcedureCall (Client& session) [Constructor]
It initializes the procedure call. It generates and sets the call identifier of the options of the procedure call. This call identifier is defined as the symbol call_id of the OpenMath Content Dictionary scscp1. The call identifier is prefixed by libSCSCP::.

ProcedureCall (void) [Destructor]
It destroys the procedure call.

9.4.1.2 set_runtimelimit

int set_runtimelimit (size_t time) [Method on ProcedureCall]
This function sets the amount of time in milliseconds, with the value time, that the SCSCP server should spend on this call. This runtime limit is defined by the symbol option_runtime of the OpenMath Content Dictionary scscp1.
On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value.

9.4.1.3 get_runtimelimit

int get_runtimelimit (size_t& time) [Method on ProcedureCall]
This function returns, in the argument time, the amount of time in milliseconds that the server should spend on this call. This amount of time is defined as the symbol option_runtime of the OpenMath Content Dictionary scscp1. If the amount of time isn’t available (not supplied by the server), the function fails.
On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value.

9.4.1.4 set_minmemory

int set_minmemory (size_t memsize) [Method on ProcedureCall]
This function sets the minimum amount of memory in bytes, with the value memsize, that the server should use on this call. This memory limit is defined by the symbol option_min_memory of the OpenMath Content Dictionary scscp1.
On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value.
9.4.1.5 get_minmemory

```c
int get_minmemory (size_t& memsize) [Method on ProcedureCall]
```

This function returns, in the argument `memsize`, the minimum amount of memory in bytes that the server should use on this call. This amount of memory is defined as the symbol `option_min_memory` of the OpenMath Content Dictionary `scscp1`. If the amount of memory isn’t available (not supplied by the server), the function fails.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value.

9.4.1.6 set_maxmemory

```c
int set_maxmemory (size_t memsize) [Method on ProcedureCall]
```

This function sets the maximum amount of memory in bytes, with the value `memsize`, that the SCSCP server should use on this call. This memory limit is defined by the symbol `option_max_memory` of the OpenMath Content Dictionary `scscp1`.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value.

9.4.1.7 get_maxmemory

```c
int get_maxmemory (size_t& memsize) [Method on ProcedureCall]
```

This function returns, in the argument `memsize`, the maximum amount of memory in bytes that the server should use on this call. This amount of memory is defined as the symbol `option_max_memory` of the OpenMath Content Dictionary `scscp1`. If the amount of memory isn’t available (not supplied by the server), the function fails.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value.

9.4.1.8 set_debuglevel

```c
int set_debuglevel (int debuglevel) [Method on ProcedureCall]
```

This function sets the debug level, with the value `debuglevel`, that the client is interested. This debug level is defined by the symbol `option_debuglevel` of the OpenMath Content Dictionary `scscp1`.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value.

9.4.1.9 get_debuglevel

```c
int get_debuglevel (int& debuglevel) [Method on ProcedureCall]
```

This function returns, in the argument `debuglevel`, the debug level that the client is interested. This debug level is defined as the symbol `option_max_memory` of the OpenMath Content Dictionary `scscp1`. If the debug level isn’t available (not supplied by the server), the function fails and returns 0.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value.

9.4.1.10 set_runtime

```c
int set_runtime (size_t time) [Method on ProcedureCall]
```

This function sets the amount of time in milliseconds, with the value `time`, that the server spent on this call. This amount of time is defined as the symbol `info_runtime` of the OpenMath Content Dictionary `scscp1`.
On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value.

9.4.1.11 get_runtime

```
int get_runtime (size_t& time) [Method on ProcedureCall]
```
This function returns, in the argument `time`, the amount of time in milliseconds that the server spent on this call. This amount of time is defined as the symbol `info_runtime` of the OpenMath Content Dictionary `scscp1`. If the amount of time isn’t available (not supplied by the server), the function fails.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value.

9.4.1.12 set_memory

```
int set_memory (size_t memsize) [Method on ProcedureCall]
```
This function sets the amount of memory in bytes, with the value `memsize`, that the server used on this call. This amount of memory is defined as the symbol `info_memory` of the OpenMath Content Dictionary `scscp1`.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value.

9.4.1.13 get_memory

```
int get_memory (size_t& memsize) [Method on ProcedureCall]
```
This function returns, in the argument `memsize`, the amount of memory in bytes that the server used for this call. This amount of memory is defined as the symbol `info_memory` of the OpenMath Content Dictionary `scscp1`. If the amount of memory isn’t available (not supplied by the server), the function fails.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value.

9.4.1.14 set_message

```
int set_message (const char *buffer) [Method on ProcedureCall]
```
This function sets the information message, with the value `buffer`, of the options of the procedure return. This information message is defined as the symbol `info_message` of the OpenMath Content Dictionary `scscp1`. The argument `buffer` can’t be NULL and won’t be duplicated by this function. So `buffer` can’t be destroyed until the instance of the object is destroyed.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value.

9.4.1.15 get_message

```
int get_message (const char * &buffer) [Method on ProcedureCall]
```
This function returns, in the argument `buffer`, the information message of the options of the procedure return. This information message is defined as the symbol `info_message` of the OpenMath Content Dictionary `scscp1`. The returned string mustn’t be modified.

On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value.
9.4.1.16 set_encodingtype

```cpp
int set_encodingtype (SCSCP_encodingtype encodingtype) {
    // Method on ProcedureCall
    This function sets the current encoding of the OpenMath objects used by the SCSCP server or the client, to send the data. The provided Openmath buffers, such as for the call sendstr, and the streams operations must use the same encoding.
    The default encoding for the SCSCP client and server is the SCSCP_encodingtype_xml.
    On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value.
}
```

9.4.1.17 get_encodingtype

```cpp
int get_encodingtype (SCSCP_encodingtype& encodingtype) {
    // Method on ProcedureCall
    This function returns, in the argument encodingtype, the current encoding of the OpenMath objects used by the SCSCP server or the client to send the data.
    On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value.
}
```

9.4.1.18 get_returntype

```cpp
int get_returntype (SCSCP_option_return& returntype) {
    // Method on ProcedureCall
    This function returns, in the argument returntype, the return type of the "procedure call" message that the server should send. The possible value of returntype are described in the function SCSCP_co_set_returntype.
    If the return type isn’t available (not supplied by the client), the function fails.
    On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value.
}
```

9.4.1.19 get_callid

```cpp
int get_callid (const char*& callid) {
    // Method on ProcedureCall
    This function returns, in the argument callid, the call id of the procedure call. The returned string mustn’t be modified.
    On exit, it returns 0 if an error occurs, otherwise the return value is a non-zero value.
}
```

9.4.2 Client::Computation

9.4.2.1 Client::Computation Constructor

```cpp
Client::Computation (Client& session): public ProcedureCall {
    // Constructor
    It initializes the procedure call on the client side. It generates and sets the call identifier of the options of the procedure call. This call identifier is defined as the symbol call_id of the OpenMath Content Dictionary scscp1. The call identifier is prefixed by libSCSCP:.
}
```

```cpp
~Client::Computation (void) {
    // Destructor
    It destroys the procedure call on the client side.
}
```
9.4.2.2 send

```cpp
int send (const char *openmathbuffer, size_t lenbuffer, SCSCP_option_return returntype)
```

The client sends a "procedure call" message to the SCSCP server with the options specified before. The server will return a result (object, cookie, nothing) depending on the value `returntype`. The array `openmathbuffer` is the argument of the procedure call and must be a valid OpenMath Application object with the same encoding as the instance. `lenbuffer` specifies the number of valid bytes in the array `openmathbuffer`.

On exit, if an error occurs, then it returns 0 and an exception is raised if the exception mechanism is enabled. Otherwise the return value is a non-zero value.

```cpp
int send (SCSCP_option_return returntype, std::ostream *&outstream) throw (Exception)
```

The client prepares a "procedure call" message to the SCSCP server with the options specified before. The server will return a result (object, cookie, nothing) depending on the value `returntype`.

The OpenMath data must be written to the returned stream `outstream`. This stream only accepts data already encoded in the OpenMath format. The application is responsible to encode data in the same encoding as specified by `get_encodingtype`.

The written data must be an OpenMath Application object. So it must start with `<OMA>` and finish with `</OMA>` if the XML encoding is used.

The "procedure call" message must be completed with a call to `finish()` or discarded with a call to `discard()`. These functions will delete the pointer `outstream`.

On exit, if an error occurs, then it returns 0 and an exception is raised if the exception mechanism is enabled. Otherwise the return value is a non-zero value.

The following example sends the "Procedure Call" message in order to compute 10!

```cpp
Client::Computation mytask(client);
std::ostream* mystream;
mytask.send(SCSCP_option_return_object, mystream);
*mystream << "<OMA>";
*mystream << "<OMS cd="integer1" name="factorial" />";
*mystream << "<OMI>10</OMI>";
mytask.finish();
```

```cpp
int send (SCSCP_option_return returntype, ostream* &outstream) throw (Exception)
```

The client prepares a "procedure call" message to the SCSCP server with the options specified before. The server will return a result (object, cookie, nothing) depending on the value `returntype`.

The data must be written to the returned stream `outstream`. The data will be converted to the OpenMath encoding using the encoding specified by `get_encodingtype`.

The written data must be an OpenMath Application object. So it must start with `beginOMA()` and finish with `endOMA()`.
The "procedure call" message must be completed with a call to `finish()` or discarded with a call to `discard()`. These functions will delete the pointer `outstream`.

On exit, if an error occurs, then it returns 0 and an exception is raised if the exception mechanism is enabled. Otherwise the return value is a non-zero value.

The following example sends the "Procedure Call" message in order to compute 10!

```cpp
Client::Computation mytask(client);
Unfstream *mystream;
mytask.send(SCSCP_option_return_object, mystream);
*mystream << beginOMA;
*mystream << OMS("integer1", "factorial") << 10;
*mystream << endOMA;
mytask.finish();
```

### 9.4.2.3 `recv`

```cpp
int recv (SCSCP_msgtype& msgtype, char*& openmathbuffer, size_t& lenbuffer)
   [Method on Client::Computation]
```

The client waits for the answer of a "procedure call" message from the server. It reads the attribute, the type and the content of the message returned by the server in response of a procedure call.

On exit, the argument `msgtype` contains the message type returned by the server.

On exit, the argument `openmathbuffer` contains the content of the message returned by the server. This array must be freed by the system call `free`. `lenbuffer` contains the number of valid bytes in the array `openmathbuffer`.

On exit, if an error occurs, then it returns 0 and an exception is raised if the exception mechanism is enabled. Otherwise the return value is a non-zero value.

```cpp
int recv (SCSCP_msgtype& msgtype, Iunfstream* &stream) throw (Exception)
   [Method on Client::Computation]
```

The client waits for the answer of a "procedure call" message from the server. It reads the attribute, the type and the content of the message returned by the server in response of a procedure call.

On exit, the argument `msgtype` contains the message type returned by the server.

On exit, the argument `stream` contains a pointer to a stream. It allows to parse the input stream. This pointer is valid until to the next call to `recv` or `send`.

On exit, if an error occurs, then it returns 0 and an exception is raised if the exception mechanism is enabled. Otherwise the return value is a non-zero value.

### 9.4.2.4 `discard`

```cpp
int discard (void) throw (Exception)
   [Method on Client::Computation]
```

It discards the "Procedure call" message previously created with `send`. The end of the message isn’t sent to the server. So the server won’t process this procedure call.

The stream, returned by the function `send`, must not be longer used after that call.

On exit, if an error occurs, then it returns 0 and an exception is raised if the exception mechanism is enabled. Otherwise the return value is a non-zero value.
9.4.2.5 finish

```cpp
int finish (void) throw (Exception)     // Method on Client::Computation
It completes the "Procedure call" message previously created with send. The end of
the message is sent to the server. So the server will process it.
The stream, returned by the function send, must not be longer used after that call.
On exit, if an error occurs, then it returns 0 and an exception is raised if the exception
mechanism is enabled. Otherwise the return value is a non-zero value.
```

9.4.3 Server::Computation

9.4.3.1 Server::Computation Constructor

```cpp
Server::Computation (IncomingClient& session) : public ProcedureCall
It initializes the instance to handle the procedure call on the server side.
```

```cpp
~Server::Computation (void)  // Destructor
It destroys the instance to handle the procedure call on the server side.
```

9.4.3.2 recv

```cpp
int recv (SCSCP_msgtype& msgtype, char*& openmathbuffer, size_t& lenbuffer) throw (Exception)
It waits for an incoming message. When a new message is available, then it reads the
attribute, the type and the content of the message sent by the client incomingclient.
On exit, the argument msgtype must be SCSCP_msgtype_ProcedureCall or SCSCP_msgtype_Interrupt. The client sends only "Procedure Call" or "Interrupt" message.
On exit, if the argument msgtype is SCSCP_msgtype_Interrupt, the call identifier of the interrupted procedure call could be retrieved with a call to get_callid().
On exit, the argument openmathbuffer contains the content of the message sent by
the client. This string must be freed by the system call free.
On exit, if an error occurs, then it returns 0 and an exception is raised if the exception
mechanism is enabled. Otherwise the return value is a non-zero value.
```

```cpp
int recv (SCSCP_msgtype& msgtype, Iunfstream*& stream) throw (Exception)
It waits for an incoming message. When a new message is available, then it reads the
attribute, the type and the content of the message sent by the client incomingclient.
On exit, the argument msgtype must be SCSCP_msgtype_ProcedureCall or SCSCP_msgtype_Interrupt. The client sends only "Procedure Call" or "Interrupt" message.
On exit, if the argument msgtype is SCSCP_msgtype_Interrupt, the call identifier of the interrupted procedure call could be retrieved with a call to get_callid().
On exit, the argument stream contains a pointer to a stream. It allows to parse the
input stream. This pointer is valid until to the next call to recv or send.
On exit, if an error occurs, then it returns 0 and an exception is raised if the exception
mechanism is enabled. Otherwise the return value is a non-zero value.
```
9.4.3.3 sendcompleted

These methods allow to send a "procedure completed" message to the client.

```cpp
int sendcompleted (const char* [Method on Client::Computation]
    openmathbuffer=NULL, size_t lenbuffer=0) throw (Exception)
```

It sends a "procedure completed" message using an openmath buffer to the client with the options specified before. The array `openmathbuffer` is the argument of the procedure call and must be a valid OpenMath object with the same encoding as the instance. If the answer is empty, `openmathbuffer` could be NULL. `lenbuffer` specifies the number of valid bytes in the array `openmathbuffer`.

On exit, if an error occurs, then it returns 0 and an exception is raised if the exception mechanism is enabled. Otherwise the return value is a non-zero value.

```cpp
int sendcompleted (std::ostream *& [Method on Client::Computation]
    outstream) throw (Exception)
```

The server prepares to send a "procedure completed" message using an openmath buffer to the client with the options specified before.

The OpenMath data must be written to the returned stream `outstream`. This stream only accepts data already encoded in the OpenMath format. The application is responsible to encode data in the same encoding as specified by `get_encodingtype`.

The written data must be an OpenMath object.

The answer must be completed with a call to `finish()`. That function will delete the pointer `outstream`.

On exit, if an error occurs, then it returns 0 and an exception is raised if the exception mechanism is enabled. Otherwise the return value is a non-zero value.

```cpp
int sendcompleted (Ounfstream*& [Method on Server::Computation]
    outstream) throw (Exception)
```

The server prepares to send a "procedure completed" message using an openmath buffer to the client with the options specified before.

The data must be written to the returned stream `outstream`. The data will be converted to the OpenMath encoding using the encoding specified by `get_encodingtype`.

The written data must be an OpenMath object.

The answer must be completed with a call to `finish()`. That function will delete the pointer `outstream`.

On exit, if an error occurs, then it returns 0 and an exception is raised if the exception mechanism is enabled. Otherwise the return value is a non-zero value.

9.4.3.4 sendterminated

That method allows to send a "procedure terminated" message to the client.

```cpp
int sendterminated (const char * cdname, [Method on Client::Computation]
    const char * symbolname, const char * message) throw (Exception)
```

It sends a "procedure terminated" message to the SCSCP client with the options specified before. The symbol of the OpenMath Error is defined by its name `symbolname` and its CD `cdname`. `message` is the message that will be inserted in a OMSTR OpenMath object.
On exit, if an error occurs, then it returns 0 and an exception is raised if the exception mechanism is enabled. Otherwise the return value is a non-zero value.

9.4.3.5 finish

```cpp
int finish (void) throw (Exception) { [Method on Server::Computation]
  It completes the answer previously created with sendcompleted. The end of the message is sent to the client. So the client will process it.
  The stream, returned by the function sendcompleted, must not be longer used after that call.
  On exit, if an error occurs, then it returns 0 and an exception is raised if the exception mechanism is enabled. Otherwise the return value is a non-zero value.
```

9.5 Stream

9.5.1 Ounfstream

This output stream handles the writing of the basic C data-types, such as int, double, .... It supports also the manipulators to build complex OpenMath objects. It encodes in binary or XML the OpenMath objects according to the current encoding of the stream. The following example builds the OpenMath Object, corresponding to $1+n^2$ :

```cpp
Ounfstream& stream =...;
stream << beginOMA << OMS("arith1","plus")
   stream << 1
   stream << beginOMA << OMS("arith1","power") << OMV("n") << 2 << endOMA
stream << endOMA;
```

9.5.1.1 beginOMA

```cpp
beginOMA (Ounfstream&) throw (Exception) { [Method on Ounfstream&]
  It writes the beginning of the structured Open Math object <OMA> to the stream stream.
  On exit, if an error occurs, then an exception is raised if the exception mechanism is enabled.
```

9.5.1.2 endOMA

```cpp
beginOMA (Ounfstream&) throw (Exception) { [Method on Ounfstream&]
  It writes the end of the structured Open Math object </OMA> to the stream stream.
  On exit, if an error occurs, then an exception is raised if the exception mechanism is enabled.
```

9.5.1.3 beginOMATP

```cpp
beginOMATP (Ounfstream&) throw (Exception) { [Method on Ounfstream&]
  It writes the beginning of the structured Open Math object <OMATP> to the stream stream.
  On exit, if an error occurs, then an exception is raised if the exception mechanism is enabled.
```
9.5.1.4 endOMATP

endOMATP (Ounfstream&) throw (Exception) [Method on Ounfstream&]

It writes the end of the structured Open Math object </OMATP> to the stream.

On exit, if an error occurs, then an exception is raised if the exception mechanism is enabled.

9.5.1.5 beginOMATTR

beginOMATTR (Ounfstream&) throw (Exception) [Method on Ounfstream&]

It writes the beginning of the structured Open Math object <OMATTR> to the stream.

On exit, if an error occurs, then an exception is raised if the exception mechanism is enabled.

9.5.1.6 endOMATTR

endOMATTR (Ounfstream&) throw (Exception) [Method on Ounfstream&]

It writes the end of the structured Open Math object </OMATTR> to the stream.

On exit, if an error occurs, then an exception is raised if the exception mechanism is enabled.

9.5.1.7 beginOME

beginOME (Ounfstream&) throw (Exception) [Method on Ounfstream&]

It writes the beginning of the structured Open Math object <OME> to the stream.

On exit, if an error occurs, then an exception is raised if the exception mechanism is enabled.

9.5.1.8 endOME

endOME (Ounfstream&) throw (Exception) [Method on Ounfstream&]

It writes the end of the structured Open Math object </OME> to the stream.

On exit, if an error occurs, then an exception is raised if the exception mechanism is enabled.

9.5.1.9 beginOMOBJ

beginOMOBJ (Ounfstream&) throw (Exception) [Method on Ounfstream&]

It writes the beginning of the structured Open Math object <OMOBJ> to the stream.

On exit, if an error occurs, then an exception is raised if the exception mechanism is enabled.
9.5.1.10 endOMOBJ

endOMOBJ (Ounfstream&) throw (Exception)  [Method on Ounfstream&]  
It writes the end of the structured Open Math object </OMOBJ> to the stream.
On exit, if an error occurs, then an exception is raised if the exception mechanism is enabled.

9.5.1.11 operator<<

Ounfstream& operator << (Ounfstream& out, int x)  [operator<< on Ounfstream]
throw (Exception)
It writes x to the stream out as the basic OpenMath object <OMI>...</OMI> according to the current encoding.
On exit, if an error occurs, then an exception is raised if the exception mechanism is enabled.

Ounfstream& operator << (Ounfstream& out, double x)  [operator<< on Ounfstream]
throw (Exception)
It writes x to the stream out as the basic OpenMath object <OMF dec="..."/> according to the current encoding.
On exit, if an error occurs, then an exception is raised if the exception mechanism is enabled.

Ounfstream& operator << (Ounfstream& out, const char * str)  [operator<< on Ounfstream]
throw (Exception)
It writes the C-style string str to the stream out as the basic OpenMath object <OMSTR>...</OMSTR> according to the current encoding.
On exit, if an error occurs, then an exception is raised if the exception mechanism is enabled.

Ounfstream& operator << (Ounfstream& out, OMF x)  [operator<< on Ounfstream]
throw (Exception)
It writes the OpenMath float x to the stream out according to the current encoding.
On exit, if an error occurs, then an exception is raised if the exception mechanism is enabled.

Ounfstream& operator << (Ounfstream& out, OMI x)  [operator<< on Ounfstream]
throw (Exception)
It writes the OpenMath integer x to the stream out according to the current encoding.
On exit, if an error occurs, then an exception is raised if the exception mechanism is enabled.

Ounfstream& operator << (Ounfstream& out, OMR x)  [operator<< on Ounfstream]
throw (Exception)
It writes the OpenMath reference x to the stream out according to the current encoding.
On exit, if an error occurs, then an exception is raised if the exception mechanism is enabled.
Ounfstream& operator << (Ounfstream& out, OMS x) throw (Exception)
   It writes the OpenMath symbol x to the stream out according to the current encoding.
   On exit, if an error occurs, then an exception is raised if the exception mechanism is enabled.

Ounfstream& operator << (Ounfstream& out, OMV x) throw (Exception)
   It writes the OpenMath variable x to the stream out according to the current encoding.
   On exit, if an error occurs, then an exception is raised if the exception mechanism is enabled.

9.5.2 Iunfstream

9.5.2.1 omtype

This type is an enumeration of the possible values of the node.

The possible values are

- SCSCP_omtype_OMI
  OpenMath integer.

- SCSCP_omtype_OMF
  OpenMath IEEE floating point number.

- SCSCP_omtype_OMV
  OpenMath variable.

- SCSCP_omtype_OME
  OpenMath symbol.

- SCSCP_omtype_OMSTR
  OpenMath string.

- SCSCP_omtype_OMB
  OpenMath byte array.

- SCSCP_omtype_OMFOREIGN
  OpenMath foreign.

- SCSCP_omtype_OMA
  OpenMath application.

- SCSCP_omtype_OMBIND
  OpenMath binding.

- SCSCP_omtype_OMATTR
  OpenMath attribution.

- SCSCP_omtype_OME
  OpenMath error.

- SCSCP_omtype_OMATP
  OpenMath attribute pair.
SCSCP_omtype_OMOBJ
OpenMath object.

SCSCP_omtype_OMBVAR
OpenMath variables used in binding.

SCSCP_omtype_OMR
OpenMath reference.

SCSCP_omtype_CONTENT
content node.

9.5.2.2 iterator_attr

This type in an iterator on the attributes of the Openmath elements. It could be used in a similar as iterator of the STL.

```cpp
Iunfstream& stream = .... ;

for (Iunfstream::iterator_attr attr = stream.get_attr(); attr.end() ; ++attr)
{
    PRINTAB(tab+1);
    cout << "attribute : \"" << attr.get_name() << ",\" = \"" << attr.get_value() <<\
         ",\"<endl;
}
```

This type defines the following methods.

bool end () const [Method on Iunfstream::iterator_attr]
It returns true if no more attributes are available, otherwise it returns false.

void operator++ () [Method on Iunfstream::iterator_attr]
It goes to the next attributes.

const char *get_name () const [Method on Iunfstream::iterator_attr]
It returns the name of the current attribute.

const char *get_value () const [Method on Iunfstream::iterator_attr]
It returns the value of the current attribute.

9.5.2.3 eof

bool eof () const [Method on Iunfstream]
It returns true if no more nodes are available in the stream.

9.5.2.4 get_attr

Iunfstream::iterator_attr get_attr () [Method on Iunfstream]
It returns an iterator on the attributes of the nodes.
9.5.2.5 get_type

Iunfstream::omtype get_type () const  
It returns as an enumeration the current node, e.g, SCSCP::omtype::OMA for an OpenMath Application. It could return SCSCP::omtype::CONTENT if the node contains only a content.

9.5.2.6 get_typename

const char* get_typename () const  
It returns as a C-style string the name of the current node, e.g, "OMA" for an OpenMath Application. It could return NULL if the node contains only a content.

9.5.2.7 get_content

const char* get_content () const  
It returns as a C-style string the content of the current node. It returns NULL if the content of the OpenMath object is empty.

9.5.2.8 beginOM

Iunfstream beginOM ()  
It returns a stream to parse the children nodes of the derived OpenMath objects.

9.5.2.9 operator>>

Iunfstream& operator >> (int& x) throw (Exception)  
It reads an OpenMath object from the stream and converts it to the integer x. If this OpenMath object can’t be converted to a floating-point, an exception is raised.
On exit, if an error occurs, then an exception is raised if the exception mechanism is enabled.

Iunfstream& operator >> (double& x) throw (Exception)  
It reads an OpenMath object from the stream and converts it to the floating-point x. If this OpenMath object can’t be converted to a floating-point, an exception is raised.
On exit, if an error occurs, then an exception is raised if the exception mechanism is enabled.

Iunfstream& operator >> (const char*& x) throw (Exception)  
It reads an OpenMath string from the stream and store it to x. If this OpenMath object can’t be converted to a string, an exception is raised.
On exit, if an error occurs, then an exception is raised if the exception mechanism is enabled.

Iunfstream& operator >> (OMF x) throw (Exception)  
It reads the OpenMath float x from the stream.
On exit, if an error occurs, then an exception is raised if the exception mechanism is enabled.

\texttt{Iunfstream\& operator >> (OMI\ x) throw (Exception)} \hspace{1cm} [operator\>>\ on \ Iunfstream]

It reads the OpenMath integer \( x \) from the stream.

On exit, if an error occurs, then an exception is raised if the exception mechanism is enabled.

\texttt{Iunfstream\& operator >> (OMR\ x) throw (Exception)} \hspace{1cm} [operator\>>\ on \ Iunfstream]

It reads the OpenMath reference \( x \) from the stream.

On exit, if an error occurs, then an exception is raised if the exception mechanism is enabled.

\texttt{Iunfstream\& operator >> (OMS\ x) throw (Exception)} \hspace{1cm} [operator\>>\ on \ Iunfstream]

It reads the OpenMath symbol \( x \) from the stream.

On exit, if an error occurs, then an exception is raised if the exception mechanism is enabled.

\texttt{Iunfstream\& operator >> (OMV\ x) throw (Exception)} \hspace{1cm} [operator\>>\ on \ Iunfstream]

It reads the OpenMath variable \( x \) from the stream.

On exit, if an error occurs, then an exception is raised if the exception mechanism is enabled.

\section*{9.6 Exception}

On errors, the \C++\ functions raise an exception of type \texttt{Exception} which derivates from the STL \texttt{std::exception}.

\subsection*{9.6.1 Constructor}

\texttt{Exception (const SCSCP\_status\*\ status) : public std::exception} \hspace{1cm} [Constructor]

It initializes the SCSCP exception wit the specified status information.

\texttt{~Exception (void)} \hspace{1cm} [Destructor]

It destroys the exception.

\subsection*{9.6.2 what}

\texttt{const char \* what () throw()} \hspace{1cm} [Method on \texttt{Exception}]

It returns a C-style character string describing the general cause of the current error.

\section*{9.7 OpenMath objects}

These OpenMath objects are usefull to perform the input or output from/to the stream. All these OpenMath objects derivate from the base class \texttt{OMBase}.

The C-style string accepted by their constructor aren’t duplicated and must exist during the life of the instance of the object.

The method \texttt{beginOM} is a common method for derived OpenMath object, such as OMA, OME, ....
9.7.1 OMBase

const char * get_id () const [Method on OMBase]
It returns the reference id (<OM .... id="...">) of the Openmath object. It may return NULL if the reference id of the object is unset.

9.7.2 OMA

OMA (const char *id=NULL) [Constructor]
It creates the OpenMath application with the reference id. paramid

Iunfstream beginOM () [Method on OMA]
It returns a stream to parse the children nodes of the derived OpenMath objects.

9.7.3 OMBIND

OMBIND (const char *id=NULL) [Constructor]
It creates the OpenMath binding with the reference id. paramid

Iunfstream beginOM () [Method on OMBIND]
It returns a stream to parse the children nodes of the derived OpenMath objects.

9.7.4 OME

OME (const char *id=NULL) [Constructor]
It creates the OpenMath error with the reference id. paramid

Iunfstream beginOM () [Method on OME]
It returns a stream to parse the children nodes of the derived OpenMath objects.

9.7.5 OMF

OMF (const char *value, const char *id=NULL) [Constructor]
It creates the OpenMath float with the value and the reference id. paramid The C-style string href isn’t duplicated. value must be in base 10.

const char * get_value () const [Method on OMF]
This function returns the value of the Openmath float.

int get_base () const [Method on OMF]
This function returns the base number of the Openmath float.

9.7.6 OMI

OMI (const char *value, const char *id=NULL) [Constructor]
It creates the OpenMath integer with the value and the reference id. paramid The C-style string href isn’t duplicated. value must be in base 10.

const char * get_value () const [Method on OMI]
This function returns the value of the Openmath integer.
9.7.7 OMR

OMR (const char* href)  [Constructor]
It creates the OpenMath reference href. The C-style string href isn’t duplicated.

    const char* get_reference() const   [Method on OMR]
    This function returns the value of the Openmath reference.

9.7.8 OMS

OMS (const char* cdname, const char* symbolname, const char* id=NULL)  [Constructor]
It creates the OpenMath symbol symbolname of the Content Dictionary cdname. The C-style strings cdname and symbolname aren’t duplicated. id is the id of this object for the future reference (see OMR). id could be NULL if unset.

    const char* get_cdname() const   [Method on OMS]
    This function returns the cd name of the Openmath symbol.

    const char* get_symbolname() const   [Method on OMS]
    This function returns the symbol name of the Openmath symbol.

9.7.9 OMV

OMV (const char* name, const char* id=NULL)  [Constructor]
It creates the OpenMath variable name. The C-style string name isn’t duplicated. id is the id of this object for the future reference (see OMR). id could be NULL if unset.

    const char* get_name() const   [Method on OMS]
    This function returns the name of the Openmath variable.
10 Design a SCSCP C++ server

The file `examples/decodeserverxx.cpp` shows the server which decodes each node of the OpenMath expression received from the client. It sends an answer to the client depending on the call options.

A simple SCSCP C++ server could be done with the following operations. This server supports the scscp versions "1.3".

```
using namespace SCSCP;

• Initialize the server
  Server server("MYCAS", "1.0", "myid");
• Listen for incoming client
  server.listen();
• Loop over new clients
  IncomingClient *incomingclient;
  while ((incomingclient=server.acceptclient())!=NULL)
  {
    • Receive the "procedure call" message: 2 solutions
      Server::Computation mytask(incomingclient);
      • solution 1: read the header and parse the openmath stream
        Iunfstream *mystream;
        mytask.recv(msgtype, mystream);
        ...
      • solution 2: read the header and store the content in a string buffer
        char *openmathbuffer;
        mytask.recv(msgtype, openmathbuffer, lenbuffer);
        ....
        free(openmathbuffer);
    • Send the answer: procedure completed or terminated?
      • Send a "procedure completed" message: 3 solutions
        • solution 1: C-style byte array using the XML or binary encoding (C-string is XML)
          const char *answer="<OMI>10</OMI>";
          mytask.sendcompleted(answer, ::strlen(answer));
        • solution 2: STL ostream using the XML encoding only
          std::ostream* & stream;
          mytask.sendcompleted(stream);
          *stream << "<OMI>10</OMI>";
          mytask.finish();
        • solution 3: unformatted stream using the XML or binary encoding
  }
```
Unfstream* stream;
mytask.sendcompleted(stream);
*stream << 10;
mytask.finish();

• Send a "procedure terminated" message
  const char *messageerror="can’t store an object";
  mytask.sendterminated("sccsp1", "error_system_specific",
                        messageerror);

• Close the connection
  delete incomingclient;
}

• Stop to listen for incoming clients
  server.close();
11 Design a SCSCP C++ client

The file examples/simplestclientxx.cpp shows the simplest client which stores the value 6177887 on the server using the C-byte array and prints the answer of the server.

The file examples/simplestclientstreamfmtxx.cpp shows the simplest client which stores the value 6177887 on the server using the STL ostream and prints the answer of the server.

The file examples/simplestclientstreamunfxx.cpp shows the simplest client which stores the value 435 on the server using the unformatted stream and prints the answer of the server.

A simple SCSCP client could be done with the following operations. This simple client will connect with the SCSCP server located on "localhost" and listening on port 26133.

- Initialize the client
  ```cpp
  SCSCP::Client client;
  ```
- Open the Connection
  ```cpp
  client.connect("localhost");
  ```
- Send a procedure call : 3 solutions Client::Computation mytask(myclient);
  - solution 1 : C-style byte array using the XML or binary encoding (C-string is XML)
    ```cpp
    const char *cmd = "<OMA><OMS cd="scscp2" name="get_allowed_heads"/></OMA>";
    mytask.send(cmd, ::strlen(cmd), SCSCP_option_return_object);
    ```
  - solution 2 : STL ostream using the XML encoding only
    ```cpp
    std::ostream*& stream;
    mytask.send(SCSCP_option_return_object, stream);
    *stream << "<OMA>";
    *stream << "<OMS cd="scscp2" name="get_allowed_heads"/>
    *stream << </OMA>";
    mytask.finish();
    ```
  - solution 3 : unformatted stream using the XML or binary encoding
    ```cpp
    ofstream* stream;
    mytask.send(SCSCP_option_return_object, stream);
    *stream << beginOMA;
    *stream << OMS("scscp2","get_allowed_heads");
    *stream << endOMA;
    mytask.finish();
    ```
- Receive the answer of the procedure call : 2 solutions
  - solution 1 : read the header and parse the openmath stream
    ```cpp
    ifstream* mystream;
    mytask.recv(msgtype, mystream);
    ...
• solution 2: read the header and store the content in a string buffer

```c
char *buffer;
size_t lenbuffer;
SCSCP_msgtype msgtype;
mytask.recv(msgtype, buffer, lenbuffer);
```

• Close the connection

```c
client.close();
```
12 References

- Symbolic Computation Software Composability Protocol (SCSCP) specification
  Version 1.3, 2009
  S.Freundt, P.Horn, A.Konovalov, S.Linton, D.Roozemond.
  http://www.symcomp.org/scscp
- OpenMath content dictionary scscp1
  D. Roozemond
  http://www.win.tue.nl/SCIEnce/cds/scscp1.html
- OpenMath content dictionary scscp2
  D. Roozemond
  http://www.win.tue.nl/SCIEnce/cds/scscp2.html