

C Scripting Language

Reference Manual V4.4.0

Author: [Peter Koch](#), [IBK](#)

Table of Contents

<u>1 Introduction</u>	1
<u>1.1 At a glance</u>	1
<u>1.2 Supported systems</u>	1
<u>1.3 License</u>	2
<u>2 Installation</u>	7
<u>2.1 Windows & OS/2 layout</u>	8
<u>2.2 Unixish systems layout</u>	8
<u>3 Using the CSL executive</u>	11
<u>3.1 Jump start</u>	11
<u>3.2 Command line parameters</u>	11
<u>3.3 Embedding CSL in batch files</u>	12
<u>4 Language</u>	15
<u>4.1 Comments</u>	15
<u>4.2 Numbers</u>	15
<u>4.3 Literals</u>	15
<u>4.4 Var and const</u>	16
<u>4.4.1 Arrays</u>	17
<u>4.4.2 Initialization</u>	17
<u>4.4.3 Static and extern</u>	18
<u>4.4.4 Dynamic relocation</u>	18
<u>4.5 Operators</u>	19
<u>4.6 Statements and blocks</u>	20
<u>4.7 Program flow</u>	21
<u>4.7.1 if</u>	21
<u>4.7.2 switch</u>	21
<u>4.7.3 while</u>	21
<u>4.7.4 do</u>	21
<u>4.7.5 for</u>	22
<u>4.8 Trace facility</u>	22
<u>4.9 Exception handling</u>	23
<u>4.9.1 Between CSL and C</u>	24
<u>4.9.2 Between CSL and C++</u>	25
<u>4.10 Functions</u>	26
<u>4.10.1 Static and forward</u>	27
<u>4.11 Predefined identifiers</u>	27
<u>5 Directives</u>	29
<u>5.1 #include</u>	29
<u>5.2 #list</u>	29
<u>5.3 #loadLibrary</u>	30
<u>5.4 #loadScript</u>	30
<u>5.5 #logFile</u>	30
<u>5.6 #if / #else / #endif</u>	31
<u>5.7 #message</u>	31
<u>5.8 #error</u>	31
<u>5.9 #block</u>	31
<u>6 System library</u>	33
<u>6.1 sysCommand</u>	34
<u>6.2 sysDate</u>	34

Table of Contents

6.3 sysDateFormat	34
6.4 sysDateTime	35
6.5 sysDirectory	35
6.6 sysElapsed	35
6.7 sysEnvVar	36
6.8 sysLoadScript	36
6.9 sysLoadLibrary	37
6.10 sysLog	37
6.11 sysLogFile	38
6.12 sysLogLevel	38
6.13 sysPrompt	39
6.14 sysSleep	39
6.15 sysSecondsSince	39
6.16 sysShow	39
6.17 sysStartDate	40
6.18 sysStartDateTime	40
6.19 sysStartTime	40
6.20 sysStartTimestamp	40
6.21 sysTime	41
6.22 sysTimestamp	41
6.23 sysTrace	41
7 String library	43
7.1 strAscii	43
7.2 strBuildRecord	43
7.3 strCenter	45
7.4 strChange	45
7.5 strChar	45
7.6 strExport	45
7.7 strFormatNumber	46
7.8 strImport	46
7.9 strIndexOf	46
7.10 strIsInteger	47
7.11 strIsNumber	47
7.12 strLastIndexOf	47
7.13 strLeftJustify	47
7.14 strLength	48
7.15 strLower	48
7.16 strParseRecord	48
7.17 strRemoveWords	49
7.18 strRightJustify	50
7.19 strSplitConnectString	50
7.20 strSplitPath	50
7.21 strSpread	51
7.22 strStrip	51
7.23 strStripExtension	51
7.24 strStripLeading	51
7.25 strStripTrailing	52
7.26 strSubString	52
7.27 strUpper	52
7.28 strWordCount	52
7.29 strWords	53

Table of Contents

8 Math library.....	55
8.1 abs.....	55
8.2 acos.....	55
8.3 asin.....	55
8.4 atan.....	55
8.5 ceil.....	56
8.6 cos.....	56
8.7 cosh.....	56
8.8 exp.....	56
8.9 floor.....	56
8.10 log.....	56
8.11 log10.....	56
8.12 max.....	56
8.13 min.....	57
8.14 pow.....	57
8.15 sqrt.....	57
8.16 sin.....	57
8.17 sinh.....	57
8.18 tan.....	57
8.19 tanh.....	57
9 Regular expression lib.....	59
9.1 Basic matching rules.....	59
9.2 Additional syntax specs.....	61
9.3 Order of precedence.....	62
9.4 rexClose.....	63
9.5 rexMatch.....	63
9.6 rexOpen.....	64
9.7 Sample.....	64
10 File library.....	67
10.1 fileClose.....	67
10.2 fileCopy.....	68
10.3 fileDelete.....	68
10.4 fileDelDir.....	68
10.5 fileEof.....	68
10.6 fileFlush.....	69
10.7 fileInfo.....	69
10.8 fileLocate.....	69
10.9 fileMakeDir.....	69
10.10 fileOpen.....	70
10.11 fileRead.....	70
10.12 fileReadLine.....	71
10.13 fileReadPos.....	71
10.14 fileRename.....	71
10.15 fileTempName.....	71
10.16 fileWrite.....	72
10.17 fileWriteLine.....	72
10.18 fileWritePos.....	72
11 Database library.....	73
11.1 daxCheckCursor.....	73
11.2 daxCommit.....	73

Table of Contents

11.3 daxConnect	73
11.4 daxDatabase	74
11.5 daxDisconnect	74
11.6 daxDispose	74
11.7 daxDone	74
11.8 daxFetch	75
11.9 daxLiteral	75
11.10 daxParse	75
11.11 daxRollback	75
11.12 daxRowsProcessed	75
11.13 daxSelectColumnName	76
11.14 daxSelectColumns	76
11.15 daxSelectColumnSize	76
11.16 daxSelectColumnType	76
11.17 daxSimple	76
11.18 daxSupply	76
11.19 Sample 1 (toys.csl)	77
11.20 Sample 2 (portable.csl)	78
11.21 Sample 3 (unknown.csl)	80
12 Async Communication	83
12.1 comBits	83
12.2 comBps	83
12.3 comClose	84
12.4 comFlow	84
12.5 comInputChars	84
12.6 comOpen	85
12.7 comParity	85
12.8 comPurgeInput	86
12.9 comRead	86
12.10 comReadChar	86
12.11 comReadTimeout	87
12.12 comStops	87
12.13 comWaitForOutput	87
12.14 comWrite	87
13 Registry/Profile handling	89
13.1 prfClose	89
13.2 prfDeleteKey	89
13.3 prfDeleteValue	90
13.4 prfGetKeys	90
13.5 prfGetValue	90
13.6 prfGetValues	90
13.7 prfKeyExists	90
13.8 prfOpen	91
13.9 prfOpenSystem	91
13.10 prfOpenUser	91
13.11 prfPath	92
13.12 prfSetValue	92
13.13 prfValueExists	92
13.14 prfValueType	92
14 Windows control	95

Table of Contents

14.1 winActivate	96
14.2 winClose	96
14.3 winCapsLock	96
14.4 winFind	96
14.5 winHide	97
14.6 winIsMaximized	97
14.7 winIsMinimized	97
14.8 winIsVisible	97
14.9 winMaximize	97
14.10 winMinimize	97
14.11 winNumLock	98
14.12 winPostText	98
14.13 winPostVKey	98
14.14 winPrintScreen	98
14.15 winRestore	99
14.16 winScrollLock	99
14.17 winShow	99
14.18 Sample (notepad.csl)	99
15 C API	103
15.1 Embedding CSL	103
15.2 Writing libraries	106
15.3 API reference	108
15.3.1 ZCslAddFunc	108
15.3.2 ZCslAddVar	108
15.3.3 ZCslCall	108
15.3.4 ZCslCallEx	109
15.3.5 ZCslClose	109
15.3.6 ZCslGet	109
15.3.7 ZCslGetError	110
15.3.8 ZCslGetResult	111
15.3.9 ZCslLoadLibrary	111
15.3.10 ZCslLoadScriptFile	112
15.3.11 ZCslLoadScriptMem	112
15.3.12 ZCslOpen	112
15.3.13 ZCslSet	113
15.3.14 ZCslSetError	113
15.3.15 ZCslSetResult	114
15.3.16 ZCslSetTraceMode	114
15.3.17 ZCslShow	115
15.3.18 ZCslStartDateTime	115
15.3.19 ZCslTrace	116
15.3.20 ZCslTraceMode	116
15.3.21 ZCslVarExists	116
15.3.22 ZCslVarResize	116
15.3.23 ZCslVarSizeof	117
16 C++ Class Interface	119
16.1 Embedding CSL	119
16.2 Writing libraries	121
16.3 Class interface reference	122
16.3.1 ZCsl constructor	123
16.3.2 ZCsl destructor	123

Table of Contents

16.3.3 ZCsl::addFunc	123
16.3.4 ZCsl::addVar	123
16.3.5 ZCsl::call	124
16.3.6 ZCsl::callEx	124
16.3.7 ZCsl::get	124
16.3.8 ZCsl::loadLibrary	125
16.3.9 ZCsl::loadScript	125
16.3.10 ZCsl::set	125
16.3.11 ZCsl::setTraceMode	126
16.3.12 ZCsl::show	126
16.3.13 ZCsl::startDateTime	127
16.3.14 ZCsl::trace	127
16.3.15 ZCsl::traceMode	127
16.3.16 ZCsl::varExists	127
16.3.17 ZCsl::varResize	127
16.3.18 ZCsl::varSizeof	127
17 CSL links	129

1 Introduction

C Scripting Language (CSL) is a well structured and easy to learn *script programming language* available for 32 bit Windows, OS/2 and Unix style systems. CSL follows the C syntax very closely and programmers used to C, C++ and JAVA will immediately be familiar with it. CSL is used like an interpreter: You write the program with your favorite editor and run it directly like any shell script.

A powerful set of libraries covers topics like system, strings, maths, files, async communications, regular expressions, registry and profiles, window control and others. The *data access library* enables high performance tasks against ORACLE, DB2, MySQL and ODBC like data import/export, schema setup scripts and other.

More than that, the CSL scripting engine can be embedded in applications. CSL has 2 programming interfaces: The C API can be used with almost every 32 bit C/C++ compiler, while the C++ class library is available for selected C++ compilers only.

1.1 At a glance

- C syntax, very familiar to C, C++ and JAVA programmers
- Sophisticated error handling by exceptions
- Comprehensive standard libraries included (system, strings, maths, files, regular expressions and more)
- Interfaces ORACLE, DB2, MySQL and every ODBC source
- Automate regular tasks or build application benchmarks with the *windows control* library
- Develop libraries to meet your special requirements using your favorite C/C++ compiler
- Embed the compact CSL scripting engine into your applications enhancing them with a powerful macro language
- Write CGI programs for your web server

1.2 Supported systems

CSL is currently tested for:

- Windows 95/98/ME (execution only)
- Windows NT/2000/XP
- OS/2
- FreeBSD
- Linux

There may be new platforms tested since this manual was updated. Check the CSL homepage at <http://csl.sourceforge.net>.

Most unixish systems with gcc /gmake will also be able to successfully compile CSL (probably with minimal changes).

Please send a note to info@ibk-software.ch if you tested CSL on a platform not yet listed here or on the CSL homepage.

1.3 License

Copyright © 1998–2001, Informatik–Buero Koch (IBK) – Landquart – Switzerland

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License version 2 as published by the Free Software Foundation.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License below for more details.

The General Public License (GPL) Version 2, June 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc. 675 Mass Ave, Cambridge, MA 02139, USA. Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:
- a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
 - b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
 - c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:
 - a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
 - b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
 - c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.
5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.
6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.
7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous

contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.
9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

2 Installation

Windows and OS/2

If you received CSL on diskettes, insert the first diskette and change to that drive. If you downloaded CSL, unzip the package into a temporary directory and change to it.

Run the installation program (for Windows SETUP.EXE, for OS/2 INSTALL.EXE) and follow the directions. After installation the temporary directory may be deleted.

Special Note for OS/2:

When using good old *pkunzip2.exe* to unpack the distribution archive, you have to rename it to an 8.3 name and use the `-d` switch to retain the directory structure:

```
ren csl-x.y.z.os2.zip csl.zip
pkunzip2 -d csl
install
```

Source distributions for unixish systems

Unpack the archive as:

```
tar xzvf csl-x.y.z.tar.gz
```

(Replace x.y.z by the version of CSL).

The archive will unpack into a single subdirectory named `csl-x.y.z`. Change to `csl-x.y.z/Unix/Gnu` and read the file *INSTALL* for further instructions how to build and install.

Binary distributions for unixish systems

You must be *root* to install the binary distribution!

- Unpack the downloaded archive `csl-x.y.z-sys-arch.tar.gz`.
- Change to the directory `csl-x.y.z-sys-arch`.
- Unpack the archive files `files.tar` to the desired destination. (The standard destination is `/usr/local`)
- Run `ldconfig`

Example:

```
tar xzf csl-4.3.0-FreeBSD-i386.tar.gz
cd csl-4.3.0-FreeBSD-i386
tar -C /usr/local -xf files.tar
ldconfig
```

NOTE:

You may drop the install directory after installation, it is not required for using csl:

```
cd ..
rm -rf csl-4.3.0-FreeBSD-i386
```

2.1 Windows & OS/2 layout

Directory	Opsys	Package	Description
.	Any	Base	Release notes, license files and installation utility files. (Don't remove the installation utility files or the CSL uninstall facility will be broken).
.\CslPath	Any	Base	Public .csl files (headers for standard libraries)
.\csldoc .\img	Any	Docs	This manual in HTML format.
.\Include	Any	API	Header files for C/C++
.\Samples	Any	Samples	CSL script samples
.\Samples\Class	Any	API samples	Samples how to use C++ class interface to embed CSL in applications and to write dll's for CSL.
.\Samples\Api	Any	API samples	Samples how to use the C-API to embed CSL in applications and to write dll's for CSL.
.\Source	Any	Source	CSL source files
.\Win\Bin .\Os2\Bin	Win32 OS/2	Base	Executables and dynamic link libraries.
.\Win\Lib .\Os2\Lib	Win32 OS/2	API	Static and import libraries for API linking
.\Win\Borland	Win32	Source	Makefiles to compile CSL with Borland C++ 5.5
.\Win\Ibm .\Os2\Ibm	Win32 OS/2	Source	Makefiles to compile CSL with VisualAge C++
.\Win\Microsoft	Any	Source	Makefiles to compile CSL with Microsoft Visual C++ V5.0

. is the root of the selected installation path

2.2 Unixish systems layout

Directory	Description
<prefix>/bin	Executables. This directory must be included in the environment variable PATH if not in a standard location like /usr/bin or /usr/local/bin.
<prefix>/lib	Shared libraries. This directory must be included in the environment variable LD_LIBRARY_PATH if not in a standard location like /usr/lib or /usr/local/lib.
<prefix>/include	Include files for C/C++
<prefix>/share/csl	Public .csl files (headers for standard libraries). This directory should be included in the environment variable CSLPATH if you want to use the headers.
<prefix>/share/csl/samples	Sample scripts
<prefix>/share/csl/samples/Class	Samples how to use C++ class interface to embed CSL in applications and to write shared libraries for CSL.
<prefix>/share/csl/samples/Api	Samples how to use the C-API to embed CSL in applications and to write shared

	libraries for CSL.
<prefix>/share/doc/csl/html	The CSL manual (this documentation). Start reading with <i>index.html</i> . It is a good idea to bookmark the CSL manual in KDE for quick access.
<prefix>/share/doc/csl/img	

<prefix> is the target directory you installed CSL to, usually /usr/local.

3 Using the CSL executive

CSL comes along with an executive program allowing to run standalone-scripts.

Syntax:

```
csl scriptfile [additional parameters]
```

The scriptfile usually has the extension *.csl* (or *.bat*, *.cmd* or none for some special purpose explained later). If the extension is one of the above, it may be omitted on the command line. If your scriptfile has another extension, it has to be provided explicitly.

The additional parameters depend on your scriptfile.

NOTE: To learn how to embed CSL in your application read the API-section.

3.1 Jump start

Start out writing a small test program with your favorite text editor and save it to a file named *hello.csl*:

```
#loadLibrary 'ZcSysLib'

main()
{
    syslog("hello world");
}
```

Now run it by entering this command on the command line:

```
csl hello
```

NOTE: You can find all samples of this manual in the *Samples* subdirectory too. You might find it however better to manually enter the smaller samples because you will learn also by typo errors in your script.

3.2 Command line parameters

Csl first saves all command line parameters into a global array named *mainArgVals*. Then it loads the script given as parameter 1 and calls the function named *main*.

Check it out by writing a new program *args.csl*:

```
#loadLibrary 'ZcSysLib'

main()
{
    for (var i=0; i<sizeof(mainArgVals); i++)
        syslog(mainArgVals[i]);
}
```

Run this sample with the command:

```
csl args my name is fred
```

The output will look like:

```
csl
args
my
name
is
fred
```

3.3 Embedding CSL in batch files

Typing *csl* in front of the script name becomes annoying sometimes. One solution to this may be to write a batch file / shell script which in turn will call *csl*. But CSL has also a feature to embed the script into the batch file itself.

Windows and OS/2

If a file starts with '@', CSL will skip the first line and start compiling in the second line. Rename your file *args.csl* from our previous example to *args.bat* (Windows) or *args.cmd* (OS/2) and modify it this way:

```
@goto exec

#loadLibrary 'ZcSysLib'

main()
{
    for (var i=0; i<sizeof(mainArgVals); i++)
        sysLog(mainArgVals[i]);
}

/*
:exec
@csl %0 %1 %2 %3 %4 %5 %6 %7 %8 %9
@rem */
```

You may now run the sample either with or without *csl* in front:

```
csl args my name is fred
...or...
args my name is fred
```

The output will however be the same in both cases:

```
csl
args
my
name
is
fred
```

There is nothing magic about this. But what is actually going on?

Since the extension of the file is *.bat* or *.cmd* respectively, the system runs it as a normal batch script by the command processor. The command processor in turn reads the first line and jumps down behind the *:exec* label where it calls the *csl* executive. The *@rem* line at the end hides the *csl* comment close from the command processor.

For CSL this file looks as a perfect source too since it ignores the first line starting with '@'.

Unixish systems

If a file starts with '#!', CSL will skip the first line and start compiling in the second line. Rename your file *args.csl* from our previous example to *args* and modify it this way:

```
#!/usr/bin/env csl
```

```
#loadLibrary 'ZcSysLib'

main()
{
    for (var i=0; i<sizeof(mainArgVals); i++)
        syslog(mainArgVals[i]);
}
```

Now you must tell the operating system, that *args* is an executable file:

```
chmod 755 args
```

You may now run the sample either with or without *csl* in front:

```
csl args my name is fred
...or...
args my name is fred
```

The output will however be the same in both cases (whereas the *args* line might also be *./args* depending on the shell):

```
csl
args
my
name
is
fred
```


4 Language

The CSL language is very close to C. However there are differences; the most significant are:

- The one and only variable type is *var* which (likely to REXX variables) may hold any number or string. No pointers, struct's and typedef's. (struct functionality is however possible by arrays)
- No goto's.
- Exception handling by try/catch/throw, fully interoperable with C++ ZException's. (Throw an exception in C++ and catch it in CSL or vice versa.
- Dynamic array allocation and reallocation.

The differences are more detailed explained in the following sections.

4.1 Comments

Any characters between `/*` and `*/` are ignored by CSL. CSL also ignores the rest of a source line behind `//`

```
/* this comment
   may spread over
   several lines */

// this comment is on a single line
```

4.2 Numbers

Numbers may be noted either decimal, octal or hexadecimal. While decimal notation can be used for integers and floating points, octal and hexadecimal notation allows integers only.

Numbers starting with `0x` or `0X` are considered hexadecimal, numbers starting with `0` are assumed to be octal and numbers starting with `1` to `9` are interpreted decimal.

Floating point constants may be written either in fixed point or exponential notation.

Examples:

```
123          // decimal integer
123.456      // decimal floating point
1.23456e+2   // same value in exponential notation
0177        // octal value
0x1f7       // hex value
```

4.3 Literals

String literals may be enclosed in single or double quotes:

```
'sandra bullock'
"bruce willis"
```

Several consecutive literals are concatenated to a single string:

```
"john's" ' car'
```

evaluates to exactly the same as

"john's car"

Literals must end before the end of line, so use the concatenation feature for long literals having to be spread over several lines.

Encoding Nonprintable Chars

Nonprintable characters are noted with a backslash followed by an encoding character or number:

Written	Description
<code>\e</code>	escape (ESC)
<code>\t</code>	horizontal tab (HT)
<code>\r</code>	carriage return (CR)
<code>\n</code>	newline/line feed (LF)
<code>\f</code>	form feed (FF)
<code>\\</code>	backslash (\)
<code>\'</code>	single quote (')
<code>\"</code>	double quote (")
<code>\ooo</code>	octal character code (<i>o</i> standing for octal digits 0..7)
<code>\xhh</code>	hex character code (<i>h</i> standing for digits 0..9 and a..f)

4.4 Var and const

There is only one single variable type in CSL: *var*

A *var* may hold any number or string. Additionally there may be a *const* qualification; however you don't write *const var* in CSL but simply *const*.

Examples:

```
var x, y;
const xMax = 80, yMax = 25;
```

While *var*'s may change contents during program execution, *const*'s cannot be changed after declaration.

var's and *const*'s may be defined anywhere in CSL, as known by C++:

```
foo()
{
  const x = 7;
  for (var k=1; k < x; k++) {
    var y = k * 3;
    sysLog(y);
  }
}
```

4.4.1 Arrays

CSL supports arrays with up to 16 dimensions. Since a *var* may hold any text or number, an array can take some functionality of structs:

```
const adr[3][2] = {
  { 1, 'fred' },
  { 2, "john" },
  { 3, 'jane' }
};
```

Local arrays (e.g. declared within a function) are allocated at runtime and may have variable indexes. This is a unique feature of CSL you won't find in C/C++:

```
var x = 12;

var aa[++x]; // aa holds 13 elements
var bb[x*2]; // bb holds 26 elements
```

Global arrays must be declared with constant indexes. This restriction is imposed since global *var*'s and *const*'s are allocated at compile time.

4.4.2 Initialization

var's may be initialized optionally, if initialization is omitted they are initialized to an empty string:

```
var x; // equivalent to var x = '';
var yMax = 12, message = 'hello world';
```

const initialization is mandatory (except for externals).

When initializing an array with a single value, all elements are set to that value:

```
var a1[100]; // all elements initialized to ''
const a2[12] = 100; // all elements set to 100
```

Array elements may be initialized similar to in C/C++ supplying the values in braces. There is a difference between global array initialization and array initialization within functions. The latter is dynamic while globals are static.

Example:

```
var adr[4][2] = {
  { 1, 'fred' },
  { 2, "john" },
  { 3 }
};
```

If *adr* is a global array it's size is truly 4 by 2 as noted in the indexes. Not explicitly initialized elements are set to an empty string so *adr* will in fact show up like:

```
1   'fred'
2   'john'
3   ''
''  ''
```

However if *adr* were a local *var* (declared within a function) the indexes are irrelevant; the number of dimensions and their size is determined by the data itself. That will be an array of 3 rows and 2 columns in this case:

```
1   'fred'
2   'john'
3   ''
```

Since the array indexes are irrelevant for initialized local arrays you may omit them at all:

```
var adr = {
  { 1, 'fred' },
  { 2, 'john' },
  { 3, }
};
```

4.4.3 Static and extern

Global identifiers may be prefixed by *static* or *extern*:

```
static var max = 100;
```

Static globals as *max* are visible only to the module they are defined in. Unlike C/C++, static's may not be defined *within* functions.

```
extern const size;
```

Since *size* is defined external there is no initializer allowed in this context. In fact you can use *extern* also as forward declaration for an identifier declared later in the same module. As long as the value of *size* is unknown you cannot use it as index when defining a global array. However you may use it to define arrays within functions:

```
var zz[size]; // illegal, unknown at compile time

var bee(var y)
{
  var kk[y][size]; // ok, allocated at runtime
} // bee
```

External declarations are necessary when using identifiers that are loaded at runtime (see *sysLoadScript* and *sysLoadLibrary*). Usually you will have the external definitions in a .csl file bound in by *#include* or *#loadScript*. Identifiers loaded at compile-time need no forward declaration if they are loaded before use.

4.4.4 Dynamic relocation

One smart feature of CSL is it's ability to dynamically reallocate variables by the *resize* statement:

```
var adr = {
  { 1, 'john' },
  { 2, 'fred' }
};
...
// now we need some more adr rows:
resize adr[10][2];
...
```

The array is reallocated dynamically so you can insert another 8 addresses. There are some things you must be aware of when resizing arrays:

Alignment

If you change another index than the highest your data will no longer be aligned in columns and rows as before:

```
resize adr[2][3];
```

Will change the table to:

```
1      'john'    2
'fred' ''       ''
```

Dimensions

The number of dimensions cannot be changed. All the following resizes are **invalid**:

```
resize adr[2][3][4];
resize adr[7];
resize adr;
```

Performance

Resizing is no cheap operation. When the total size of the array grows, a new array is allocated internally and the values have to be copied. Avoid tons of small resize's and do the job in bigger chunks to avoid performance problems.

Arguments

Resizing references is possible when the referenced variable is unindexed:

```
foo(var x)
{
    syslog('resizing from '+sizeof(x)+' to 7');
    resize x[7];
}

main()
{
    var a[3], b[5][4];
    foo(a); // ok, because unindexed.
    foo(b[2]); // runtime error thrown in foo!
}
```

4.5 Operators

Operators in descending precedence:

Operators	Comments
() []	
! + - ++ -- & sizeof exists	Unary pre/postfixes. & is only allowed in parameter lists
* / \ %	\ is integer divide
- +	With + one or both operands may be a string (non-numeric) whereby string concatenation is performed instead of a numeric add. is the string concatenation operator.
< <= > >=	If one or both operands are strings, a literally string-compare will take place.
== !=	If one or both operands are strings, a literally string-compare will take place.
	Evaluation of A B: In CSL, B will always be evaluated. In C/C++, B will only be evaluated if A is true.
	Evaluation of A B: In CSL, B will always be evaluated. In C/C++, B will only be evaluated if A is false.

<code>= += -= *= /= \=</code> <code>%= =</code>	With += source or target may be a string (non-numeric) whereby string concatenation is performed instead of a numeric add.
<code>,</code>	Comma is used to group several expressions into a single statement.

There are no bitwise (`| & ^^ ~~`) and no conditional (`? :`) operators in CSL.

Expressions true/false evaluation:

- if the expression is a number, zero values are false and nonzero values are true.
- if the expression is no number, empty strings are false and nonempty are true.

The *sizeof* operator returns 1 for simple var's. For arrays it returns the number of elements for the given index:

```
var xy[5][4];

sizeof(xy); // 20
sizeof xy[1]; // 4
sizeof xy[2][3]; // 1
```

The *exists* operator checks if a variable exists and returns 1 or 0. Within directives the condition is met if the var/const is declared despite the fact indexes may be invalid. In script code the var/const must be allocated and any indexes must be valid to return 1:

```
extern const a[];

#message exists maxSize // 1, even if only declared but
                        // not yet implemented
const a[3] = { 1, 2, 3 };

#message exists(maxSize[99]) // still 1, indexes are not checked!

foo()
{
  syslog(exists a); // 1
  syslog(exists(a[1])); // 1
  syslog(exists a[99]); // 0, indexes are checked in code!
}
```

4.6 Statements and blocks

An expression becomes a statement when it is followed by a semicolon:

```
var x = 5;
y = ++x * 7;
foo();
```

Braces `{` and `}` are used to group statements into a compound statement, so they are syntactically equivalent to a single statement:

```
{ y=z; foo(); }
```

The comma may be used to group several expressions into a single statement too:

```
y=z, foo();
```

4.7 Program flow

This chapter describes the program flow constructs to make decisions and process loops.

4.7.1 if

```
if (expression) statement-1 [else statement-2]
```

If *expression* evaluates true *statement-1* is executed, otherwise *statement-2*. The *else*-part is optional.

4.7.2 switch

```
switch (expression-0)
{
  case expression-1: [statements]
  [case expression-2: [statements]]
  ...
  [case expression-N: [statements]]
  [default: statements]
}
```

expression-0 is compared with *expression-1* ... *expression-N*. As soon as a match is found, all subsequent *statements* within the switch block are executed. The statements after *default* are executed if none of the previous expressions was matched.

Within the statements, *break* is used to leave the switch block premature.

Note that unlike C, *expression-1* ... *expression-N* don't have to be constants.

4.7.3 while

```
while (expression) statement
```

While *expression* evaluates true *statement* is executed. (The check is done before *statement* is executed).

The loop may be left premature by *break*. With *continue* the rest of *statement* is skipped and the loop continues with the next *expression* check.

4.7.4 do

```
do statement while (expression)
```

statement is repeated until *expression* becomes false. (The check is done after *statement* execution).

The loop may be left premature by *break*. With *continue* the rest of *statement* is skipped and the loop continues with the next *expression* check.

4.7.5 for

`for ([expression-1]; [expression-2]; [expression-3]) statement`

expression-1 is executed once as initializer. As long as *expression-2* evaluates true *statement* and *expression-3* are executed in sequence. If *expression-2* is not present, it is taken as permanently true.

The loop may be left premature by *break*. With *continue* the rest of *statement* is skipped and the loop continues with the next *expression-3* followed by the *expression-2* check.

4.8 Trace facility

The trace facility allows tracing of p-code (the interpreted meta-code), function and block entry/exit, and expressions/messages. Trace output is sent to stderr and so can be redirected to a file in case. Trace facility is made up by a set of elements:

- The *trace* statement is used to monitor expressions and show messages within script code. The API [ZCslTrace](#) and the class member [ZCsl::trace](#) enable C and C++ code to display diagnostic messages in a similar way.
- The [#block](#) directive names code blocks for tracing.
- The [sysTrace](#) function controls if, and what kind of information is traced. The same can be done from the C API with [ZCslTraceMode](#) / [ZCslSetTraceMode](#) and the C++ class members [ZCsl::traceMode](#) / [ZCsl::setTraceMode](#).

Example (trace.csl):

```
#loadLibrary 'ZcSysLib'

test(const &arr[])
{
    var evens = 0, odds = 0;
    for (var i = 0; i < sizeof(arr); i++) {
        #block 'for block'
        trace 'value = '|arr[i];
        if (arr[i] % 2) {
            #block 'odd branch'
            odds++;
            trace 'odds = '|odds;
        } else {
            #block 'even branch'
            evens++;
            trace 'evens = '|evens;
        }
    }
}

main()
{
    sysTrace(sysTraceCode);
    const vals = { 3, 12, 17 };
    sysTrace(sysTraceInfo);
    test(vals);
    sysTrace(sysTraceNone);
}
```

Output when run:

```
#
#trace.csl: var main()
#
```

#address	opcode	parameter	tos	tos-1
#	3	pop	1	<stack bottom>
#	4	push	<stack bottom>	
#	5	push	vals[3]	<stack bottom>
#	6	allc	vals[3]	
#	7	push	3	<stack bottom>
#	8	push	vals[0]	<stack bottom>
#	9	storc	vals[0]	3
#	10	push	12	<stack bottom>
#	11	push	vals[1]	<stack bottom>
#	12	storc	vals[1]	12
#	13	push	17	<stack bottom>
#	14	push	vals[2]	<stack bottom>
#	15	storc	vals[2]	17
#	16	push	14	<stack bottom>
#	17	push	1	<stack bottom>
#	18	call	sysTrace	14

```

-ZcSysLib: var sysTrace([const mode])
+trace.csl: var test(const )
+for block
  >value = 3
  +odd branch
  >odds = 1
  -odd branch
-for block
+for block
  >value = 12
  +even branch
  >evens = 1
  -even branch
-for block
+for block
  >value = 17
  +odd branch
  >odds = 2
  -odd branch
-for block
-+trace.csl: var test(const )
+ZcSysLib: var sysTrace([const mode])

```

4.9 Exception handling

Exceptions are handled by *try/catch/throw* as known by C++ and JAVA:

```

foo()
{
  try {
    ....
    var exc = { 'error in foo:', 'the message' };
    throw exc;
    ....
  } // try
  catch (const xx[]) {
    for (var i = 0; i < sizeof xx; i++)
      sysLog(xx[i]);
  } // catch
} // foo

```

Unlike C++ there may only be one catch block following the try block. The reason is, that there is only one variable type in CSL and it makes no sense to define several catch blocks.

throw must be followed by an identifier name or an expression as argument. The identifier may be any simple *var/const*, or array with any number of dimensions.

catch expects an identifier followed by empty braces [] as parameter. No matter what number of dimensions the thrown value had, the caught value will be a one-dimensional array.

4.9.1 Between CSL and C

Errors emitted by *ZCslSetError* will raise an exception in the calling CSL code:

C function

```

-----
ZExportAPI(void) myFunc(ZCslHandle csl)
{
    char buf[20];
    long bufsiz;
    int argc;

    bufsiz = sizeof(buf);
    if ( ZCslGet(csl, "argCount", buf, &bufsiz) ) return;
    argc = atoi(buf);
    if (argc == 2) {
        ZCslSetError("Error in myFunc:");
        ZCslSetError("%%% argcount must be 1 or 3");
        return;
    } // if
    ...
} // myFunc

```

CSL program

```

-----
#loadLibrary 'ZcSysLib'
#loadLibrary 'ZcMyLib'

main()
{
    try { myFunc(1, 2); }
    catch (var exc[]) {
        syslog('caught exception with '+sizeof(exc)+' line(s):');
        for (var i = 0; i < sizeof exc; i++)
            syslog(exc[i]);
    } // catch
} // main

```

Output when running:

```

-----
caught exception with 2 line(s):
Error in myFunc:
%%% argcount must be 1 or 3

```

Exceptions thrown in CSL will be returned as error by *ZCslCall*:

CSL Function

```

-----
test(const mode)
{
    if (mode < 0 || mode > 3) {
        const exc[2] = {
            'error in test():',
            'invalid mode: '+mode
        };
        throw exc;
    } // if
}

```

```
....
} // test
```

C Code

```
-----
...
static char *args[] = { "5" };
errs = ZCslCall(csl, module, "test", 1, args);
if (errs) {
    // errs will be 1 and ZCslGetError(csl, 0,...)
    // will return 'invalid mode:5'
...

```

4.9.2 Between CSL and C++

It is possible to throw a ZException within a C++ function and catch it in CSL:

C++ function

```
-----
static ZString myCppFunc(ZCsl* csl)
{
    int argc = csl->get("argCount").asInt();
    if (argc == 2)
        ZTHROWEXC("%%% argcount must be 1 or 3");
    ...
} // myCppFunction
```

CSL program

```
-----
#include <ZcSysLib.dll>
#include <ZcMyLib.dll>

main()
{
    try { myCppFunc(1, 2); }
    catch (var exc[]) {
        syslog('caught exception with '+sizeof(exc)+' line(s):');
        for (var i = 0; i < sizeof exc; i++)
            syslog(exc[i]);
    } // catch
} // main
```

Output when running:

```
-----
caught exception with 1 line(s):
argcount must be 1 or 3
```

The vice versa is also possible:

CSL Function

```
-----
test(const mode)
{
    if (mode < 0 || mode > 3) {
        const exc[2] = {
            'error in test():',
            'invalid mode:'+mode
        };
        throw exc;
    }
}
```

```

} // if
....
} // test

```

C++ Function

```

void cppTest()
{
    try {
        ZCsl csl;
        csl.loadScript("test.csl");
        ZString ret = csl.call("cppTest.exe", "test", 1, "5");
        ....
    } // try
    catch (const ZExceptionerr) {
        for (int i = err.textCount()-1; i >= 0; i--)
            cerr << err.text(i) << endl;
    } // catch
} // cppTest

```

4.10 Functions

Functions always have a return type of *var*. If a function does not explicitly return a value, an empty string will be returned. Since each function returns a *var*, it is not necessary to put *var* in front of the function header, although you may do so to indicate that your function ought to return something meaningful:

```

var foo()          // the 'var' is optional
{
    return 'hello';
} // foo

main()
{
    const x = foo(); // x = 'hello'
} // main

```

The parameter list may have a fixed or variable number of parameters. Optional parameters are enclosed in braces []:

```

fixParam(var a, var b)
// 2 mandatory parameters
...

varParam(var a, [var b, var c])
// 1 mandatory and 2 optional parameters
...

```

If there are optional parameters, CSL generates a local *const* named *argCount* holding the passed number of arguments. CSL will validate that the number of arguments passed isn't less than the number of mandatory parameters and also not bigger than the total number of parameters.

Your function must validate the number of optional parameters before accessing them:

```

foo(const a, [const b])
{
    if (argCount == 2)
        syslog('you passed 2 args: '+a+' and '+b);
    else
        syslog('you passed 1 arg: '+a);
}

```

If your function doesn't change the *arg* values you may define them *const* instead of *var* so they are protected by CSL.

CSL also supports parameter passing by reference with the operator & (as known by C++). In fact arrays may only be passed by reference and not by value:

```
#loadLibrary 'ZcSysLib'

groom(var& a, var b[])
{
    const maxDim = 10;
    a = 'groom';
    const dim2 = sizeof(b[0]);
    const dim1 = sizeof(b) / dim2;
    if (dim1 > maxDim || dim2 > maxDim)
        throw '%% groom was designed for dims <= '+maxDim;
    for (var y = 0; y < dim1; y++)
        for (var x = 0; x < dim2; x++)
            b[y][x] = y*dim2+x;
    return a+' done';
}

main()
{
    var aa;
    var bb[3][4][5];
    syslog(groom(aa, bb[1])); // groom done
    syslog(aa);              // groom
    syslog(bb[1][2][3]);    // 13
}
```

Array-parameters never have an explicit index as you see in the example above. If your function imposes restrictions on array size(s), it's up to your function to verify that by use of the *sizeof* operator as we did in the example above.

4.10.1 Static and forward

Forward or external definition of functions is done by defining the function header followed by a semicolon:

```
xtern(var b);
```

External declarations are necessary when using functions that are loaded at runtime (see *sysLoadScript* and *sysLoadLibrary*). Usually you will have the external definitions in a file bound in by *#include* or *#loadScript*. Functions loaded at compile-time need no forward declaration if they are loaded before use.

Static functions are visible only to the module they were defined in:

```
// calcXy is a local function not visible
// by other modules:
static var calcXy(var x, var y)
{
    ....
} // calcXy
```

4.11 Predefined identifiers

Identifier	Scope	Value	Comment
const true	global	1	
const false	global	0	
const PATHSEPARATOR	global	/ or \	Depending on operating system
const MAXLONG	global	2147483647	Useful for some library functions

<code>const cslVersion</code>	global	CSL kernel version	
<code>const cslCompiler</code>	global	IBM, BORLAND, MICROSOFT or GNU	Compiler csl kernel was built by
<code>const cslBuilt</code>	global	Date and time of kernel build	
<code>static const cslFileName</code>	global	file/module name	
<code>const argCount</code>	local	# of arguments passed to function	Only present in functions with optional parameters
<code>const cslFunction</code>	local	function name	Only present in functions
<code>const mainArgVals[]</code>	global	arguments on command line	Only present when run by csl executive

5 Directives

Directives are introduced by `#`. Unlike C and C++ directives are not processed by a precompiler but by the compiler itself. They serve as commands executed at *compile time*.

One effect of this is that you may use *const*'s and constant expressions as arguments for the directives.

5.1 #include

Includes a file at compile time.

```
#include 'ZcMathLb'
```

...or...

```
static const mathLib = 'ZcMathLb';
```

```
#include mathLib
```

If no extension is given, CSL will add the default extension `.csl`. The file is searched in the paths defined in environment variable `CSLPATH`. If no `CSLPATH` is defined the file is searched in the current working directory.

NOTE: `#include` will perform unconditionally which may be a problem when the same file is included from several sources. You'd better use `#loadScript` to load forward declaration because that will automatically avoid multiple loads (as you would do in C/C++ by a sequence of `#ifndef _XXY_ ... #define _XXY_ ... declarations ... #endif`)

5.2 #list

Enables/disables P-Code listing depending on the argument which may be `0/1` or `false/true` respectively. The listing will be displayed by `sysLog`, so using this directive requires the system library loaded in advance.

Optionally you may save the listing in a file by opening a logfile in advance:

```
#loadLibrary 'ZcSysLib' // load system library
#logFile 'listing' // open logfile
#list true // enable listing

var main()
{
    sysLog('hello world');
} // main

#logFile '' // close logfile
```

This will show the listing on screen and save it to `listing.log`:

```
test.csl: var main()

address  opcode  parameter
-----
         0  push   hello world
         1  push   1
         2  call   sysLog
         3  pop
         4  push
         5  ret

code: 36  text: 21  total: 57
```

5.3 #loadLibrary

Load a DLL during compile time. Argument is the library name:

```
#loadLibrary 'ZcSysLib'
```

... or ...

```
const x1 = 'Zc', x2 = 'SysLib';
```

```
#loadLibrary x1+x2
```

If no extension is given *.dll* is assumed on Windows and OS/2, and *.so.X* is assumed on unixish systems (where X is the major version of CSL).

OS/2:

The library is searched in all directories listed by *LIBPATH* in *CONFIG.SYS*, or in the environment variables *BEGINLIBPATH* and *ENDLIBPATH*.

Win32:

The library is searched in the directory where *CSL.EXE* is (respectively your own VisualAge C++ Program using the CSL API), the current directory, and all directories listed in the *PATH* environment variable.

Unixish systems:

The library is searched in the directories listed in the environment variable *LD_LIBRARY_PATH*.

CSL will never load the same library more than once. Any attempt to do so will be silently ignored.

5.4 #loadScript

Compiles another csl source file at compile time.

```
#loadScript 'MyFuncs'
```

... or ...

```
static const module = 'MyFuncs';
```

```
#loadScript module
```

If no extension is given, CSL will add the default extension *.csl*. The file is searched in the current working directory and in the paths defined in environment variable *CSLPATH*.

CSL will never load the same script more than once. Any attempt to do so will be silently ignored.

5.5 #logFile

Opens or closes a logfile at compile time. Useful when generating P-Code listings (see *#list*). This directive uses the system library so it has to be loaded in advance:

```
#loadLibrary 'ZcSysLib' // load system library
```

```
#logFile 'listing.lst' // open logfile
```

....

```
#logFile '' // close logfile
```

If no extension is given, CSL will add the default extension *.log*.

5.6 #if / #else / #endif

Enables conditional compilation:

```
#if const expression
  code compiled when true
[#else
  code compiled when false]
#endif
```

Useful for writing portable scripts as:

```
#loadLibrary 'ZcSysLib' // load system library

#if sysIsUnixFamily
  (Code for unixish systems)
#else
  (Code for Windows or OS/2)
#endif
```

5.7 #message

Prints a message to stdout at compile time

```
#message 'compiling foo.csl'
```

5.8 #error

Forces an error at compile time:

```
#if !sysIsLinux
  #error 'Sorry, this script requires Linux!'
#endif
```

5.9 #block

Names a code block in order to [trace](#) it. (Place this directive at the begin of the block):

```
#loadLibrary 'ZcSysLib'

main()
{
  sysTrace(sysTraceBlks+sysTraceMsgs); // turn on block+msg tracing
  if (sysIsWindows) {
    #block 'win branch'
    trace 'windows related code here';
  } else {
    #block 'others'
    trace 'other opsys related code here';
  }
}
```

Running on windows will show:

```
+win branch
>windows related code here
-win branch
```


6 System library

Files

ZcSysLib.dll / libZcSysLib.so
The implemented functions

ZcSysLib.csl
External declarations used when loading at runtime

The system library will normally be loaded at *compile-time* by *#loadLibrary* since the function *sysLoadLibrary* must be present before a library can be loaded at runtime by a CSL script.

However there may be a situation where the system library gets loaded at runtime by a C++ function and a CSL script for any reason must be loaded before. For this case you might need *ZcSysLib.csl*.

Globals

Identifier	Description
sysVersion	Current version of the library
sysCompiler	Compiler the library was built by (IBM, BORLAND, MICROSOFT, GNU)
sysBuilt	Date and time when the library was compiled
sysLibtype	Interface the library is using (API or NATIVE = C++ class interface)
sysDateFormatISO sysDateFormatEuro sysDateFormatUS	Formats for <i>sysDateFormat</i>
sysShowCallStack sysShowFunctions sysShowFullStack sysShowGlobals sysShowLibraries	Modes for sysShow
sysIsOS2 sysIsUnixFamily sysIsLinux sysIsBSD sysIsAIX sysIsSolaris sysIsUnix sysIsWindows sysIsWin95Family sysIsWinNTFamily sysIsWinServer sysIsWin95 sysIsWin98 sysIsWinME sysIsWinNT3 sysIsWinNT4 sysIsWin2000 sysIsWinXP	Platform identification 0/1

sysTraceNone sysTraceCode sysTraceFuncs sysTraceBlks sysTraceMsgs sysTraceInfo sysTraceAll	Modes for sysTrace
--	------------------------------------

6.1 sysCommand

```
sysCommand(
  const command) // system command
```

Executes any system command, batch or program through the command interpreter.

Returns the return code from the command interpreter.

Example:

```
var retcode = sysCommand('dir > dir.lst');
```

6.2 sysDate

```
sysDate()
```

Returns the current date.

Example:

```
sysLog('The current date is '+sysDate());
```

6.3 sysDateFormat

```
sysDateFormat(
  [const format]) // new date format to set
```

Query or change current date format. If no parameter is given, the current date format is queried only.

Valid date formats are:

Format	Description
sysDateFormatISO	ISO date format as <i>YYYY.MM.DD HH.MM.SS</i>
sysDateFormatEuro	European date format as <i>DD.MM.YYYY HH:MM:SS</i>
sysDateFormatUS	US date format as <i>MM/DD/YYYY HH.MM.SS</i>

Returns current date format.

Example:

```
sysDateFormat(sysDateFormatISO);
sysLog(sysDateTime);
sysDateFormat(sysDateFormatEuro);
sysLog(sysDateTime);
sysDateFormat(sysDateFormatUS);
```

```
sysLog(sysDateTime);
```

Output:

```
2001.06.20 16.00.23
20.06.2001 16:00:23
06/20/2001 16.00.23
```

6.4 sysDateTime

```
sysDateTime()
```

Returns the current date and time.

Example:

```
sysLog('The current date and time is '+sysDateTime());
```

6.5 sysDirectory

```
sysDirectory(
    [const path])    // new path if supplied
```

Query or change current working directory. If no arg is passed a query is done. If an arg is passed, the current directory is changed to the path.

Returns fully qualified current working directory in either case.

Examples:

```
sysLog('current dir is '+sysDirectory());
// May give an output as:
//   current directory is F:\IBKProj\Csl\Kernel

try {
    sysDirectory('d:\os2\boot');
}
catch (exc[]) {
    // change directory failed
    ...
}
```

6.6 sysElapsed

```
sysElapsed()
```

Returns elapsed seconds since start of CSL (ZCsl object creation time)

Example:

```
sysLog('elapsed time was '+sysElapsed()+ ' sec(s)');
```

6.7 sysEnvVar

```
sysEnvVar(
    const varname) // name of environment variable
```

Queries the value of an environment variable

Example:

```
sysLog('path is currently: '+sysEnvVar('path'));
```

6.8 sysLoadScript

```
sysLoadScript(
    const filename) // name of script file
```

Compiles and loads a script file at runtime. The script is searched in the current working directory and all directories listed in environment variable *CSLPATH*. To access any functions or identifiers in the script they have to be forward declared.

Example:

```
#loadLibrary 'ZcSysLib'

extern var foovar[]; // implemented in foo.csl
foo(var xy);        // implemented in foo.csl

main()
{
    sysLoadScript('foo'); // load foo.csl
    foovar[1] = 1234;
    foo('hello');
}
```

CSL won't load the same script more than once; any attempts to do so will be silently ignored.

You may want to put the forward declarations into a file *foo.csl* and include that file at compile time. I recommend to do this by *#loadScript* rather than *#include*, because that will avoid multiple loading. (In C/C++ you would have to make constructions like *#ifndef _FOO_ ... #define _FOO_ ... body ... #endif*)

Example:

```
#loadLibrary 'ZcSysLib'
#loadScript 'foo.csl' // forward declarations

main()
{
    sysLoadScript('foo'); // load foo.csl at runtime
    sysLoadScript("scripts\Foo.Csl"); // will be ignored since already loaded
    foovar[1] = 1234;
    foo('hello');
}
```

6.9 sysLoadLibrary

```
sysLoadLibrary(
    const dllname) // name of library
```

Loads a library (DLL or shared library) at runtime.

OS/2: The Library is searched in all directories listed by *LIBPATH* in *CONFIG.SYS*, or in the environment variables *BEGINLIBPATH* and *ENDLIBPATH*.

Win32: The Library is searched in the directory where *CSL.EXE* is (respectively your own VisualAge C++ Program using the CSL API), the current directory, and all directories listed in the *PATH* environment variable.

Unixish systems: The Library is searched all directories listed in the *LD_LIBRARY_PATH* environment variable.

To access any functions or identifiers of the library they have to be forward declared (usually by a .csl file).

Example:

```
#loadLibrary 'ZcSysLib'

static const strLib = 'ZcStrLib'

#loadScript strLib+'.csl' // include forwards

main()
{
    sysLoadLibrary(strLib); // load ZcStrLib.csl
    sysLog(strUpper('john wayne'));
}
```

NOTES:

CSL will not load the same library more than once; any attempts to do so will be silently ignored.

I recommend to load the forward declarations by *#loadScript* rather than *#include*, because that will avoid multiple includes. (In C/C++ you would have to make constructions like *#ifndef _XXX_ ... #define _XXX_ ... body ... #endif*)

6.10 sysLog

```
sysLog(
    [const message, // message to display (default: none)
     const raw,     // raw mode (default: false)
     const quiet]) // silent mode: to logfile only (default: false)
```

Displays a message on screen and writes it to the logfile if one was opened. If no argument is passed, an empty line is logged. While the message is displayed on the screen as is, each line in the logfile will start with a timestamp.

If *raw* is *true*, logging is done in raw mode, e.g. no log level indenting and no timestamp in the logfile.

If *quiet* is *true*, the message is written to the logfile only, but not displayed on the screen.

Examples:

```
sysLog(); // empty line
sysLog('message'); // message cooked mode
sysLog('test', true); // test in raw mode
sysLog('hello', 0, 1); // write to logfile only
```

6.11 sysLogFile

```
sysLogFile(
    [const filename]) // name of logfile to open
```

Opens or closes a logfile. If no argument is given the currently open logfile is closed. If an argument is given, the currently open logfile is closed and a new one is opened. The logfile is always opened in append mode.

If no extension is given, *.log* is appended.

Returns name of logfile.

Example:

```
#loadLibrary 'ZcSysLib'
#loadLibrary 'ZcStrLib'

main()
{
    // open logfile with same name as script but extension .log
    sysLogFile(strStripExtension(cslFileName));
    ...
}
```

6.12 sysLogLevel

```
sysLogLevel(
    [const level]) // value added to current level
```

Query or change current login level. If no parameter or 0 is given, the current level is queried only. If a nonzero positive or negative number is passed, its value is added to the current logging level.

The logging level ranges from 0 to 39 (default 0). Each level indents logging output by 2 spaces.

Returns current logging level.

Example:

```
sysLog('test');
sysLogLevel(+1); sysLog('test');
sysLogLevel(+1); sysLog('test');
sysLog(sysLogLevel());
sysLogLevel(-2); sysLog('test');
```

Output:

```
test
  test
    test
  2
test
```

6.13 sysPrompt

```
sysPrompt(
  [message, // message to display as prompt
   raw])   // raw logging mode if true
```

Displays a prompt and waits for user input. Prompt and entered information are saved to the logfile if one is open.

Returns the user input.

Example:

```
var x = sysPrompt('please enter your name: ');
sysLog('welcome '+x);
```

6.14 sysSleep

```
sysSleep(
  const millisecs) // # of millisecs to sleep
```

Delays program execution by the requested time in milliseconds.

Example:

```
sysSleep(3000); // delay 3 seconds
```

6.15 sysSecondsSince

```
sysSecondsSince(
  timestamp1, // first timestamp
  [timestamp2]) // second timestamp
```

Returns the # of seconds between 2 timestamps (timestamp2 – timestamp1). If timestamp2 is omitted, the current timestamp is taken for timestamp2.

Example:

```
const start = sysTimestamp(); // save start timestamp
doSomeAction();
sysLog('doSomeAction() was running for '|sysSecondsSince(start)|' seconds.');
```

6.16 sysShow

```
sysShow(
  const mode, // what to show
  [const depth]) // how much
```

Shows internal CSL information depending on the *mode* selected:

Mode	Description
sysShowFunctions	Show all functions currently loaded by CSL
sysShowCallStack	Show track of all functions currently open
sysShowFullStack	As <i>sysShowCallStack</i> , additionally show all local vars/consts of the functions.
sysShowGlobals	Show all public globals and all static globals of all modules loaded.

<code>sysShowLibraries</code>	Show all libraries currently loaded
-------------------------------	-------------------------------------

Example:

```
sysShow(sysShowFunctions);
```

6.17 sysStartDate

```
sysStartDate()
```

Return start date, e.g. date when ZCsl object was created.

Example:

```
sysLog('CSL was started on '+sysStartDate());
```

6.18 sysStartDateTime

```
sysStartDateTime()
```

Return start date and time, e.g. date and time when ZCsl object was created.

Example:

```
sysLog('CSL was started at '+sysStartDateTime());
```

6.19 sysStartTime

```
sysStartTime()
```

Return start time, e.g. time when ZCsl object was created.

Example:

```
sysLog('CSL was started at '+sysStartTime());
```

6.20 sysStartTimestamp

```
sysStartTimestamp()
```

Returns the timestamp when csl was started as *YYYY.MM.DD.HH.MM.SS*.

Example:

```
sysLog('csl was started at '+sysStartTimestamp());
```

6.21 sysTime

```
sysTime()
```

Returns the current time.

Example:

```
sysLog('current time is '+sysTime());
```

6.22 sysTimestamp

```
sysTimestamp()
```

Returns the current timestamp as *YYYY.MM.DD.HH.MM.SS*.

Example:

```
sysLog('current timestamp is '+sysTimestamp());
```

6.23 sysTrace

```
sysTrace(
  [const mode]) // sysTrace...
```

Query or set [trace](#) mode. Predefined mode constants are:

Mode	Description
sysTraceNone	Trace output is turned off (default)
sysTraceCode	Trace P-code instructions together with the 2 top stack elements.
sysTraceFuncs	Trace function entry and exit. Trace output will be indented within the function.
sysTraceBlks	Trace entry and exit of named blocks. Trace output will be indented within the block. Use #block to name blocks.
sysTraceMsgs	Trace expressions of trace statement or API ZCslTrace and ZCsl::trace member.

The mode constants above may be combined by adding them. Combined constants are predefined as:

Mode	Description
sysTraceInfo	sysTraceFuncs + sysTraceBlks + sysTraceMsgs
sysTraceAll	sysTraceCode + sysTraceFuncs + sysTraceBlks + sysTraceMsgs

Returns: current trace mode.

Example:

```
sysTrace(sysTraceCode); // turn tracing on
```

The trace output may look somewhat as:

```
test.csl: var main()
```

address	opcode	parameter	tos	tos-1
19	push	x	<stack bottom>	
20	incv		x	<stack bottom>
21	jmp	7	<stack bottom>	
7	push	x	<stack bottom>	
8	load		x	<stack bottom>
9	push	5	3	<stack bottom>
10	lss		5	3
11	jf	22	1	<stack bottom>
.

7 String library

Files

ZcStrLib.dll / libZcStrLib.so
The implemented functions

ZcStrLib.csl
External declarations used when loading at runtime

Globals

Identifier	Description
strVersion	Current version of the library
strCompiler	Compiler the library was built by (IBM, BORLAND, MICROSOFT, GNU)
strBuilt	Date and time when the library was compiled
strLibtype	Interface the library is using (API or NATIVE = C++ class interface)

7.1 strAscii

```
strAscii(  
    const character)    // character to convert
```

Returns the ASCII code of the *character*.

Example:

```
sysLog(strAscii('1')); // 49
```

7.2 strBuildRecord

```
strBuildRecord(  
    const mode,           // file mode  
    const &fmts[],       // field formats  
    const &vals[],       // field values  
    [const nullind]);    // null indicators in vals (default=false)
```

Creates an ASCII record. Typically used by database export scripts together with the *dax* and *file* libraries. These file modes are supported:

Mode	Description
A,a	ASCII fix
c	Comma separated, text in single quotes
C	Comma separated, text in double quotes
s	Semicolon separated, text in single quotes
S	Semicolon separated, text in double quotes
t	Tab separated, text in single quotes

T	Tab separated, text in double quotes
U,u	Tab separated, text unquoted

The field formats are records of 2 ... n words. The 1st word is a field type from this list:

Format	Description
t	text (ascii=left adjusted)
T	text (ascii=right adjusted)
f	fixtext (ascii=left adjusted)
F	fixtext (ascii=right adjusted)
n	number with decimal point (leading sign)
N	number with decimal point (trailing sign)
x	number with fixed fraction digits (no decimal point, leading sign)
X	number with fixed fraction digits (no decimal point, trailing sign)

The next words are used as:

- 2nd word = (maximum) width in chars
- 3rd word (numbers only) = # of fractional digits
- 3rd...nth word (fixtext only): text

The data to fill into the record are stored in *vals*. If *nullind* is true, the data is arranged in 2 columns (for example `data[totFields][2]`), where the first (index 0) holds the value and the second (index 1) is the NULL value indicator (`true/1 = value is null, false/0 = value is not null`).

Example:

```

...
const dataLayout = {
  't 20', // article #
  't 30', // name
  'n 6', // actual stock
  'f 3 EUR', // currency
  'n 8 2' // price
};
...
var fvals = {
  '1122.344.102.00',
  'Leather football champion',
  112,
  12.95
};
fileWriteLine(expFile, strBuildRecord('s', dataLayout, fvals));
fileWriteLine(expFile, strBuildRecord('a', dataLayout, fvals));
...

```

Will write this lines into the export file:

```

'1122.344.102.00';'Leather football champion';112;'EUR';12.95
1122.344.102.00      Leather football champion      000112EUR00012.95

```

7.3 strCenter

```
strCenter(
    const string,    // string to center
    const width,    // width of field
    [const padd])  // padding character (default = ' ')
```

Centers a string in a field of *width*.

Examples:

```
strCenter('hello', 11);           // '  hello  '
strCenter('hello', 11, '*');     // '***hello***'
```

7.4 strChange

```
strChange(
    const string,    // string to treat
    const oldpatt,  // old pattern to be replaced
    const newpatt,  // new pattern
    [const start,  // start position in string (default = 1)
     const count,  // # of occurrences to replace (default = all)
     const ignorecase]) // case independent search (default = false)
```

Replaces occurrences of a pattern by another text.

Examples:

```
const s = 'abc def ghi abc def';

strChange(s, 'abc', 'ABCX');      // ABCX def ghi ABCX def
strChange(s, 'abc', 'AB', 2);     // abc def ghi AB def
strChange(s, 'abc', 'ABC', 1, 1); // ABC def ghi abc def
```

7.5 strChar

```
strChar(
    const asciival) // ascii code to convert
```

Returns the character represented by *asciival*.

Example:

```
sysLog(strChar(49)); // '1'
```

7.6 strExport

```
strExport(
    const string) // string to export
```

Converts all special characters to escaped characters.

Example:

```
var x = 'john\s\tnew\tcar\x01';
sysLog(x);
sysLog(strExport(x));
```

Output:

```
john's new car
john\s\tnew\tpcar\001
```

7.7 strFormatNumber

```
strFormatNumber(
    const val,          // value to format
    const width,       // width of field
    [const frac])     // # of fractional digits (default = 0)
```

Formats a number for output. The result is a string of *width* characters with the value right adjusted. If the value does not fit into *width*, the result will be as long as the value requires at least.

Examples:

```
strFormatNumber(1.5, 10, 2); // ' 1.50'
strFormatNumber(123.456, 10, 2); // ' 123.45'
strFormatNumber(123.456, 10); // ' 123'
```

7.8 strImport

```
strImport(
    const string) // string to import
```

Converts all escaped characters to special characters.

Example:

```
var x = 'another\tbrick in the\twall';
sysLog(x);
sysLog(strImport(x));
```

Output:

```
another\tbrick in the\twall
another brick in the wall
```

7.9 strIndexOf

```
strIndexOf(
    const string,      // string to search in
    const patt,       // pattern searched
    [const start,     // start position in string (default=1)
    const ignorecase]) // case independent search (default=false)
```

Searches an occurrence of *patt* in *string*. If the pattern is found, the 1-based index of the first character is returned. If the pattern is not found, 0 is returned.

Examples:

```
strIndexOf('hello world', 'o'); // 5
strIndexOf('hello world', 'o', 6); // 8
strIndexOf('hello world', 'x'); // 0
```

7.10 strIsInteger

```
strIsInteger(
    const string) // string to test
```

Returns true if the string represents an integer (long), otherwise false.

Example:

```
strIsInteger(1.56); // 0
strIsInteger('abc'); // 0
strIsInteger(-123); // 1
```

7.11 strIsNumber

```
strIsNumber(
    const string) // string to test
```

Returns true if the string represents a number, otherwise false.

Examples:

```
strIsNumber(1.56); // 1
strIsNumber('abc'); // 0
strIsNumber(-123); // 1
```

7.12 strLastIndexOf

```
strLastIndexOf(
    const string, // string to search in
    const patt, // pattern searched
    [const start, // start position in string (default=MAXLONG)
    const ignorecase]) // case independent search (default=false)
```

Searches an occurrence of *patt* in *string* starting at end of string and going to the beginning. If the pattern is found, the 1-based index of the first character is returned. If the pattern is not found, 0 is returned.

Examples:

```
strLastIndexOf('hello world', 'o'); // 8
strLastIndexOf('hello world', 'o', 6); // 5
strLastIndexOf('hello world', 'x'); // 0
```

7.13 strLeftJustify

```
strLeftJustify(
    const string, // string to center
    const width, // width of field
    [const padd]) // padding character (default = ' ')
```

Left justifies a string in a field of *width*.

Examples:

```
strLeftJustify('hello', 11); // 'hello      '
strLeftJustify('hello', 11, '*'); // 'hello*****'
```

7.14 strLength

```
strLength(
    const string) // source string
```

Returns the length of the string in characters

Example:

```
strLength('abc'); // 3
```

7.15 strLower

```
strLower(
    const string) // string to convert
```

Converts a string to lowercase

Example:

```
strLower('Peter Koch'); // peter koch
```

7.16 strParseRecord

```
strParseRecord(
    const rec,           // record to parse
    const mode,         // file mode (1)
    const &fmts[],      // field formats (2)
    var &vals[],        // fiels values returned
    [const nullind]);   // null indicators in vals (default=false)
```

Parses an ASCII record. Typically used by database import scripts together with the *dax* and *file* libraries. These file modes are supported:

Mode	Description
A,a	ASCII fix
C,c	Comma separated
S,s	Semicolon separated
T,t	Tab separated, text in quotes
U,u	Tab separated, text unquoted

The field formats are records of 2 ... n words. The 1st word is a field type from this list:

Format	Description
T,t	text
F,f	fixtext (ignored)
n	number with decimal point (leading sign)
N	number with decimal point (trailing sign)
x	number with fixed fraction digits (no decimal point, leading sign)

X number with fixed fraction digits (no decimal point, trailing sign)

The next words are used as:

- 2nd word = (maximum) width in chars
- 3rd word (numbers only) = # of fractional digits

The data extracted from the record are stored in *vals*. If *nullind* is true, the data is arranged in 2 columns (for example `data[totFields][2]`), where the first (index 0) holds the value and the second (index 1) is the NULL value indicator (`true/1` = value is null, `false/0` = value is not null).

Example:

```

...
const dataLayout = {
  't 20',      // article #
  't 30',      // name
  'n 6',       // actual stock
  'f 3',       // currency, ignore
  'n 8 2'     // price
};
...
var fvals[4];
strParseRecord(
  "1122.344.102.00';Leather football champion';112;'USD';12.95",
  's', dataLayout, fvals)
);
// The result will be:
// fvals[0] = '1122.344.102.00';
// fvals[1] = 'Leather football champion';
// fvals[2] = 112;
// fvals[3] = 12.95;
...

```

7.17 strRemoveWords

```

strRemoveWords(
  const string, // string holding the sentence
  const start,  // starting word, index based 1
  [const count]) // # of words

```

Remove words from a string. If *count* is omitted, all remaining words are removed.

Example:

```

const s = 'the quick brown fox jumps';

strRemoveWords(s,2,2); // 'the fox jumps'
strRemoveWords(s,3);  // 'the quick'

```

7.18 strRightJustify

```
strRightJustify(
    const string,    // string to center
    const width,    // width of field
    [const padd])  // padding character (default = ' ')
```

Right justifies a string in a field of *width*.

Examples:

```
strRightJustify('hello', 11);           // '      hello'
strRightJustify('hello', 11, '*');     // '*****hello'
```

7.19 strSplitConnectionString

```
strSplitConnectionString(
    const connstr,    // connect-string
    [var& password,  // extension
     var& connection, // directory
     var& database]) // drive
```

Splits a connect string into its 4 parts *database*, *userid*, *password* and *connection*. The connect-string has the following syntax:

```
[database:][userid][/password][@connection]
```

Any of the 4 parts may be missing. The database-part is converted to uppercase; the other parts are left in original case.

Returns the userid.

Example:

```
var db, name, pass, conn;

name = strSplitConnectionString(
    'db2:Sandra/Bullock@Hollywood',
    pass, conn, db);

// db   = 'DB2'
// name = 'Sandra'
// pass = 'Bullock'
// conn = 'Hollywood'
```

7.20 strSplitPath

```
strSplitPath(
    const path,      // full file path
    [var& ext,       // extension
     var& dir,       // directory
     var& drive])   // drive
```

Splits a path name into its 4 parts *drive*, *directory*, *filename* and *extension*.

Returns the filename.

Example:

```
var file, ext, dir, drive;

file = strSplitPath('F:\\IBK\\DOC\\CSL.INF', ext, dir, drive);
```

```
// drive = 'F:'
// dir   = '\\IBK\DOC\'
// file  = 'CSL'
// ext   = '.INF'
```

7.21 strSpread

```
strSpread(
    const string)    // string to spread
```

Spreads a string by inserting a blank after each character.

Examples:

```
strSpread('hello');           // 'h e l l o '
strSpread(strSpread('bye')); // 'b y e '
```

7.22 strStrip

```
strStrip(
    const string, // string as input
    [const char]) // character to strip (default = ' ')
```

Strips leading and trailing characters from string.

Example:

```
strStrip(' abc '); // 'abc'
```

7.23 strStripExtension

```
strStripExtension(
    const filename) // file name as input
```

Removes the extension from a file name.

Example:

```
strStripExtension('c:\\os2.old\\aic7770.add'); // 'c:\\os2.old\\aic7770'
```

7.24 strStripLeading

```
strStripLeading(
    const string, // string as input
    [const char]) // character to strip (default = ' ')
```

Strips leading characters from string.

Example:

```
strStripLeading(' abc '); // 'abc '
```

7.25 strStripTrailing

```
strStripTrailing(
    const string, // string as input
    [const char]) // character to strip (default = ' ')
```

Strips trailing characters from string.

Example:

```
strStripTrailing(' abc '); // ' abc'
```

7.26 strSubString

```
strSubString(
    const string, // source string
    const start, // starting position, index based 1
    [const count, // # of characters
     const padchar]) // padding character
```

Extracts a substring from a string. If *count* is omitted, all remaining characters are taken.

Examples:

```
const s = 'hello world';

strSubString(s,2,2); // el
strSubString(s,4); // lo world
strSubString(s,1,20,'*'); // hello world*****
strSubString(' ',1,20,'-'); // -----
```

7.27 strUpper

```
strUpper(
    const string) // string to convert
```

Converts a string to uppercase

Example:

```
strUpper('Peter Koch'); // 'PETER KOCH'
```

7.28 strWordCount

```
strWordCount(
    const string) // source string
```

Returns the number of words.

Example:

```
strWordCount("where's the pizza?"); // 3
```

7.29 strWords

```
strWords(  
    const string,    // string holding the sentence  
    const start,    // starting word, index based 1  
    [const count]) // # of words
```

Extract words from a string. If *count* is omitted, all remaining words are taken.

Example:

```
const s = 'the quick brown fox jumps';  
  
strWords(s,2,2); // quick brown  
strWords(s,4);  // fox jumps
```


8 Math library

Files

ZcMathLib.dll / libZcMathLib.so

The implemented functions

ZcMathLib.csl

External declarations used when loading at runtime

Globals

Identifier	Description
mathVersion	Current version of the library
mathCompiler	Compiler the library was built by (IBM, BORLAND, MICROSOFT, GNU)
mathBuilt	Date and time when the library was compiled
mathLibtype	Interface the library is using (API or NATIVE = C++ class interface)

8.1 abs

```
var abs(const x)
```

Returns the absolute value of the argument x .

8.2 acos

```
var acos(const x)
```

Calculates the arc cosine of x , expressed in radians, in the range 0 to π . The value of x must be in the range from -1 to $+1$.

8.3 asin

```
var asin(const x)
```

Calculates the arc sine of x , expressed in radians, in the range $-\pi/2$ to $+\pi/2$. The value of x must be in the range from -1 to $+1$.

8.4 atan

```
var atan(const x)
```

Calculates the arc tangent of x , expressed in radians, in the range $-\pi/2$ to $+\pi/2$.

8.5 ceil

```
var ceil(const x)
```

Returns the smallest integer value greater or equal to x .

8.6 cos

```
var cos(const x)
```

Calculates the cosine of x . x is expressed in radians.

8.7 cosh

```
var cosh(const x)
```

Calculates the hyperbolic cosine of x . x is expressed in radians.

8.8 exp

```
var exp(const x)
```

Calculates the exponential function of x (e to the exponent x).

8.9 floor

```
var floor(const x)
```

Calculates the largest integer smaller or equal to x .

8.10 log

```
var log(const x)
```

Calculates the natural logarithm (base e) of x . x must greater than 0.

8.11 log10

```
var log10(const x)
```

Calculates the base 10 logarithm of x . x must greater than 0.

8.12 max

```
var max(const x, const y)
```

Returns the larger of the two values x and y .

8.13 min

```
var min(const x, const y)
```

Returns the smaller of the two values x and y .

8.14 pow

```
var pow(const x, const y)
```

Calculates the value of x to the power of y .

8.15 sqrt

```
var sqrt(const x)
```

Calculates the non-negative square root of x . x must be greater or equal to 0.

8.16 sin

```
var sin(const x)
```

Calculates the sine of x . x is expressed in radians.

8.17 sinh

```
var sinh(const x)
```

Calculates the hyperbolic sine of x . x is expressed in radians.

8.18 tan

```
var tan(const x)
```

Calculates the tangent of x . x is expressed in radians.

8.19 tanh

```
var tanh(const x)
```

Calculates the hyperbolic tangent of x . x is expressed in radians.

9 Regular expression lib.

Regular Expressions (REs) are used to determine if a character string of interest is matched somewhere in a set of character strings. You can specify more than one character string for which you wish to determine if a match exists.

The functions in this library use regular expressions in a similar way to the UNIX *awk*, *ed*, *grep*, and *egrep* commands.

The search for a matching sequence starts at the beginning of the string and stops when the first sequence matching the expression is found. The first sequence is the one that begins earliest in the string. If the pattern permits matching several sequences at this starting point, the longest sequence is matched.

To use a regular expression, first compile it with *rexOpen*. You can then use *rexMatch* to compare the compiled expression to several strings. When you have finished with the expression, use *rexClose* to free it from memory.

Files

ZcRexLib.dll

The implemented functions

ZcRexLib.csl

External declarations used when loading at runtime

Globals

Identifier	Description
<i>rexVersion</i>	Current version of the library
<i>rexCompiler</i>	Compiler the library was built by (IBM, BORLAND, MICROSOFT, GNU)
<i>rexBuilt</i>	Date and time when the library was compiled
<i>rexLibtype</i>	Interface the library is using (API or NATIVE = C++ class interface)
<i>rexMatchNotbol</i> <i>rexMatchNoteol</i>	Modes for <i>rexMatch</i>
<i>rexOpenExtended</i> <i>rexOpenIgnorecase</i> <i>rexOpenNewline</i> <i>rexOpenNosubreps</i>	Flags used by <i>rexOpen</i>

9.1 Basic matching rules

An ordinary character matches itself. The simplest form of regular expression is a string of characters with no special meaning. A special character preceded by a backslash matches itself. For basic regular expressions (BREs), the special characters are:

. [\ * ^ \$

For extended regular expressions (EREs), the special characters also include:

() + ? { |

(EREs are supported when you specify the *rexOpenExtended* flag in *rexOpen*.)

A period (.) without a backslash matches any single character. For EREs, it matches any character except the null character. An expression within square brackets ([]), called a bracket expression, matches one or more characters or collating elements.

Bracket Expressions

A bracket expression itself contains one or more expressions that represent characters, collating symbols, equivalence or character classes, or range expressions:

[string]

Matches any of the characters specified. For example, [abc] matches any of a, b, or c.

[^string]

Does not match any of the characters in string. The caret immediately following the left bracket ([) negates the characters that follow. For example, [^abc] matches any character or collating element except a, b, or c.

[collat_sym–collat_sym]

Matches any collating elements that fall between the two specified collating symbols, inclusive. The two symbols must be different, and the second symbol must collate equal to or higher than the first. For example [r–t] would match any of r, s, or t.

NOTE: To treat the hyphen (–) as itself, place it either first or last in the bracket expression, for example: [–rt] or [rt–]. Both of these expressions would match –, r, or t.

[[:collat_symbl:]]

Matches the collating element represented by the specified single or multicharacter collating symbol collat_symbl. For example [[:ch.]] matches the character sequence ch. (In contrast, [ch] matches c or h.)

[[:collat_symbl=]]

Matches all collating elements that have a weight equivalent to the specified single or multicharacter collating symbol collat_symbl. For example, assuming a, à, and â belong to the same equivalence class, [[:a=]] matches any of the three. If the collating symbol does not have any equivalents, it is treated as a collating symbol and matches its corresponding collating element (as for [[:.]]).

[[:char_class:]]

Matches any characters that belong to the specified character class char_class. For example, [[:alnum:]] matches all alphanumeric characters.

NOTE: To use the right bracket (]) in a bracket expression, you must specify it immediately following the left bracket ([) or caret symbol (^). For example, []x] matches the characters] and x; [^]x] does not match] or x; [x]] is not valid.

You can combine characters, special characters, and bracket expressions to form REs that match multiple characters and subexpressions. When you concatenate the characters and expressions, the resulting RE matches any string that matches each component within the RE. For example, cd matches characters 3 and 4 of the string abcde; ab[[:digit:]] matches ab3 but not abc. For EREs, you can optionally enclose the concatenation in parentheses.

9.2 Additional syntax specs

You can also use other syntax within an RE to control what it matches:

$\backslash(expression)$

Matches whatever *expression* matches. You only need to enclose an expression in these delimiters to use operators (such as `*` or `+`) on it and to denote subexpressions for back referencing (explained later in this section). For EREs, use the parentheses without the backslashes: $(subexpression)$

$\backslash n$

Matches the same string that was matched by the *n*th preceding expression enclosed in $\backslash(\backslash)$ or, for EREs, $()$. This is called a back reference. *n* can be 1 through 9. For example, $\backslash(ab \backslash) \backslash 1$ matches `abab`, but does not match `ac`. If fewer than *n* subexpressions precede $\backslash n$, the back reference is not valid.

NOTE: You cannot use back references in EREs.

*expression**

Matches zero or more consecutive occurrences of what *expression* matches. *expression* can be a single character or collating symbol, a subexpression, or a back reference (for BREs). For example, $[ab]^*$ matches `ab` and `ababab`; b^*cd matches characters 3 to 7 of `cabbbcbdeb`.

expression $\backslash\{m\}$

Matches exactly *m* occurrences of what *expression* matches. *expression* can be a single character or collating symbol, a subexpression, or a back reference (for BREs). For example, $c \backslash \{ 3 \backslash \}$ matches characters 5 through 7 of `ababccccc` (the first 3 `c` characters only). For EREs, use the braces without the backslashes: $\{m\}$

expression $\backslash\{m,\}$

Matches at least *m* occurrences of what *expression* matches. *expression* can be a single character or collating symbol, a subexpression, or a back reference (for BREs). For example, $\backslash(ab \backslash) \backslash \{ 3 , \}$ matches `abababab`, but does not match `ababac`. For EREs, use the braces without the backslashes: $\{m, \}$

expression $\backslash\{m,u\}$

Matches any number of occurrences, between *m* and *u* inclusive, of what *expression* matches. *expression* can be a single character or collating symbol, a subexpression, or a back reference (for BREs). For example, $bc \backslash \{ 1 , 3 \backslash \}$ matches characters 2 through 4 of `abccc` and characters 3 through 6 of `abbcccccc`. For EREs, use the braces without the backslashes: $\{m, u\}$

expression

Matches only sequences that match *expression* that start at the first character of a string or after a new-line character if the *rexOpenNewline* flag was specified. For example, ab matches `ab` in the string `abcdef`, but does not match it in the string `cd efab`. The expression can be the entire RE or any subexpression of it.

expression $\$$

Matches only sequences that match *expression* that end the string or that precede the new-line character if the *rexOpenNewline* flag was specified. For example, $ab\$$ matches `ab` in the string `cd efab` but does not match it in the string `abcdef`. The expression must be the entire RE.

$^expression\$$

Matches only an entire string, or an entire line if the *rexOpenNewline* flag was specified. For example, $^abcde\$$ matches only `abcde`.

In addition to those listed above, you can also use the following specifiers for EREs (they are not valid for BREs):

expression $+$

Matches what one or more occurrences of *expression* matches. For example, $a+(bc)$ matches `aaaaabc`;

`(bc)+` matches characters 1 through 6 of `bcbbcbb`.

expression?

Matches zero or one consecutive occurrences of what *expression* matches. For example, `b?c` matches character 2 of `acabbb` (zero occurrences of `b` followed by `c`).

expression/expression

Matches a string that matches either *expression*. For example, `a((bc) | d)` matches both `abc` and `ad`.

9.3 Order of precedence

Like CSL operators, the regular expression (RE) syntax specifiers are processed in a specific order. The order of precedence for basic regular expressions (BREs) and extended regular expressions (EREs) are described below, from highest to lowest. The specifiers in each category are also listed in order of precedence.

The order of precedence for regular expressions is:

Collation-related bracket symbols	<code>[==]</code> <code>[::]</code> <code>[..]</code>
Special characters	<code>\spec_char</code>
Bracket expressions	<code>[</code>
Subexpressions and back references	<code>\(\)</code> <code>\n</code>
Repetition	<code>*</code> <code>\{m\}</code> <code>\{m,\}</code> <code>\{m,n\}</code>
Concatenation	
Anchoring	<code>^</code> <code>\$</code>

The order of precedence for Extended Regular Expressions is:

Collation-related bracket symbols	<code>[==]</code> <code>[::]</code> <code>[..]</code>
Special characters	<code>\spec_char</code>
Bracket expressions	<code>[</code>
Grouping	<code>()</code>
Repetition	<code>*</code> <code>+</code> <code>?</code> <code>{m}</code> <code>{m,}</code> <code>{m,n}</code>
Concatenation	
Anchoring	<code>^</code> <code>\$</code>
Alternation	<code> </code>

9.4 rexClose

```
rexClose(          // close regular expression handle
    const handle); // rex handle
```

Use this function to free all memory associated to the RE handle when the RE processing is finished. At return the handle is no longer valid.

9.5 rexMatch

```
rexMatch(          // compile regular expression
    const handle,  // rex handle
    const string,  // string to match
    const nmatch,  // # of matches to find
    var &match[[]], // index & length of every match
    [const flags]); // match flags
```

Compares the *string* against the compiled regular expression represented by *handle* to find a match between the two.

nmatch is the number of substrings in *string* that *rexMatch* should try to match with subexpressions in *handle*. The array you supply for *match* must have at least *nmatch* by 2 elements.

rexMatch fills in the elements of the array *match* with the starting index and the length of the matched substrings. The zeroth element of the array corresponds to the entire pattern. If there are more than *nmatch* subexpressions, only the first *nmatch* - 1 are stored.

If *nmatch* is 0, or if the *rexOpenNosubreps* flag was set when *handle* was created with *rexOpen*, *rexMatch* ignores the *match* argument and only returns 0 or 1 indicating if a match was found.

flags defines customizable behavior of *rexMatch*:

Flag	Description
rexMatchNotbol	Indicates that the first character of string is not the beginning of line.
rexMatchNoteol	Indicates that the last character of string is not the end of line.

When a basic or extended regular expression is matched, any given parenthesized subexpression of the original pattern could participate in the match of several different substrings of string. The following rules determine which substrings are reported in *pmatch*:

1. If a subexpression participated in a match several times, *rexMatch* stores index and length of the last matching substring in *match*.
2. If a subexpression did not match in the source string, the number of matches returned is less than *nmatch*.
3. If a subexpression contains subexpressions, the data in *match* refers to the last such subexpression.

If the *rexOpenNosubreps* flag was set when *handle* was created by *rexOpen*, the contents of *match* are unspecified. If the *rexOpenNewline* flag was not set when *handle* was created, new-line characters will not be treated as end-of-line.

Return value

If a match is found, *rexMatch* returns the number of matches including subreports. If no match is found *rexMatch* returns 0.

9.6 rexOpen

```
rexOpen(                                // compile regular expression
    const pattern,                       // pattern to compile
    [const flags]);                     // open flags
```

Compiles the source regular expression in *pattern* into an executable version and returns a handle for subsequent calls to *rexMatch* and *rexClose*.

flags defines the attributes of the compilation process:

Flag	Description
rexOpenExtended	Support extended regular expressions.
rexOpenIgnorecase	Ignore case in match.
rexOpenNewline	Treat new-line character as a special end-of-line character; it then establishes the line boundaries matched by the ^ and \$ patterns, and can only be matched within a string explicitly using \n. (If you omit this flag, the new-line character is treated like any other character.)
rexOpenNosubreps	Ignore the number of subexpressions specified in pattern. When you compare a string to the compiled pattern (using <i>rexMatch</i>), the string must match the entire pattern. <i>rexMatch</i> then returns a value that indicates only if a match was found; it does not indicate at what point in the string the match begins, or what the matching string is.

9.7 Sample

```
#loadLibrary 'ZcSysLib'
#loadLibrary 'ZcStrLib'
#loadLibrary 'ZcRexLib'

main()
{
    const pattern = '\\(sim[a-z]le\\) \\1';
    const string = 'a very simple simple string';

    syslog('compile regular expression');
    var rx = rexOpen(pattern);
    var match[6][2];

    syslog('find and display match');
    syslogLevel(+1);
    var m = rexMatch(rx, string, 6, match);
    if (m) {
        for (var i = 0; i < m; i++)
            syslog(
                match[i][0]+' '
                +match[i][1]+' \\1'
                +strSubString(string,match[i][0], match[i][1])
                +'\n'
            );
    } else
        syslog('no match found');
    syslogLevel(-1);

    syslog('close and release mem');
    rexClose(rx);
} // main
```

When executing this sample the output will be:

```
compile regular expression
find and display match
  8 13 'simple simple'
  8 6 'simple'
close and release mem
```


10 File library

Files

ZcFileLb.dll / libZcFileLb.so

The implemented functions

ZcFileLb.csl

External declarations used when loading at runtime

Globals

Identifier	Description
fileVersion	Current version of the library
fileCompiler	Compiler the library was built by (IBM, BORLAND, MICROSOFT, GNU)
fileBuilt	Date and time when the library was compiled
fileLibtype	Interface the library is using (API or NATIVE = C++ class interface)
fileOpenRead fileOpenWrite fileOpenAppend fileOpenTrunc fileOpenOld fileOpenNew fileOpenBinary	Modes for <i>fileOpen</i>
fileStdIn fileStdOut fileStdErr	Handles for standard input, output and error. (These handles are opened automatically in text mode.)

10.1 fileClose

```
fileClose(  
    const handle)    // file handle
```

Closes a file previously opened by *fileOpen*. All open files are closed automatically at CSL termination.

Example:

```
// show contents of file test.txt  
var fh = fileOpen('test.txt', fileOpenRead);  
while (!fileEof(fh)) sysLog(fileReadLine(fh));  
fileClose(fh);
```

10.2 fileCopy

```
fileCopy(
  const source,    // source file name
  const dest)     // destination file name
```

Copies a file.

This is a binary copy ignoring any system attributes. If you'd like any system attributes copied use `sysCommand('copy ...')` instead.

Example:

```
fileCopy('config.sys', 'config.bak');
```

10.3 fileDelete

```
fileDelete(
  const filename) // name of file to delete
```

Deletes a file.

Example:

```
fileDelete('test.txt');
```

10.4 fileDelDir

```
fileDelDir(
  const dirname) // name of directory to delete
```

Deletes a directory. (Directory must be empty)

Example:

```
fileDelDir('c:\test\output');
```

10.5 fileEof

```
fileEof(
  const handle) // check for eof
```

Check if EOF has occurred during read.

Returns true if eof occurred, otherwise false.

Example:

```
// show contents of file test.txt
var fh = fileOpen('test.txt', fileOpenRead);
while (!fileEof(fh)) syslog(fileReadLine(fh));
fileClose(fh);
```

10.6 fileFlush

```
fileFlush(
  const handle) // handle of file
```

Flushes the output buffers of a file writing all pending data to device.

Example:

```
fileWrite(fileStdOut, 'Enter name: ');
fileFlush(fileStdOut); // force output
```

10.7 fileInfo

```
fileInfo(
  const filename) // name of file queried
```

Returns detailed information about a file:

- The size of the file in bytes
- The last access date/time as YYYYMMDDHHMMSS
- The last modification date/time
- The creation date/time
- The full filename

You can extract the parts with *strWords*, but take in mind there may be filenames containing spaces.

Example:

```
fileInfo('test.txt');
// '125 19980205142536 19980205122807 19980205121355 F:\IBKProj\Csl\test.txt'
```

10.8 fileLocate

```
fileLocate(
  const filename, // name of file to search
  const varname) // environment variable with path's
```

Searches a file in all directories listed in a specific environment variable as *PATH*, *INCLUDE*, *LIB* and so on. Returns fully qualified pathname of found file. If file is not found an empty string is returned.

Example:

```
var zip = fileLocate('pkunzip2.exe', 'path');
if (zip == '') throw 'pkunzip2.exe not found';
sysLog('zip = '|zip); // 'zip = f:\ibk\pbin\pkunzip2.exe'
```

10.9 fileMakeDir

```
fileMakeDir(
  const dirname) // name of directory to create
```

Creates a directory. Only one directory may be created at a time. To create a path you may have to call *fileMakeDir* several times.

Example:

```
fileMakeDir('C:\Test');
```

```
fileMakeDir('C:\Test\Files');
```

10.10 fileOpen

```
fileOpen(
  const filename, // name of file
  const mode)    // open mode
```

Opens a file for reading/writing. *Mode* is the sum of any of the following constants:

Constant	Description
fileOpenRead	Open file for reading
fileOpenWrite	Open file for writing
fileOpenAppend	Open and set write position to EOF
fileOpenTrunc	Truncate file if opened file existed before
fileOpenOld	The file ought to exist (don't combine this with fileOpenNew).
fileOpenNew	The file must not exist (don't combine this with fileOpenOld)
fileOpenBinary	Open in binary mode. By default files are opened in text mode where EOL is represented by a LF only. In Binary mode EOL may also be represented as CR LF, depending on the operating system.

Returns a file handle that must be used in subsequent calls to *fileRead*, *fileWrite*, *fileClose* etc.

Examples:

```
// open a file in text mode to append text
var fh = fileOpen('logfile.log',
  fileOpenWrite+fileOpenAppend);

// open a file in binary mode for random access
var fh = fileOpen('address.dbf',
  fileOpenRead+
  fileOpenWrite+
  fileOpenBinary);
```

10.11 fileRead

```
fileRead(
  const handle, // file handle
  [const count]) // # of chars to read. default = 1
```

Read a number of chars from file. If reading a file sequentially check for eof before each read.

Returns the character read.

Example:

```
// copy a file
var inp = fileOpen('source.dat',
  fileOpenRead+fileOpenBinary);
var out = fileOpen('target.dat',
  fileOpenWrite+fileOpenBinary);
while (!fileEof(inp))
  fileWrite(out, fileRead(inp, 512));
fileClose(inp);
```

```
fileClose(out);
```

10.12 fileReadLine

```
fileReadLine(
    const handle) // file handle
```

Reads a line from file. For best results the file should not be opened in binary mode (otherwise there may be cr's at the end of the returned text).

Check for eof before reading.

Returns the line (the terminating LF character is removed).

Example:

```
// show contents of file test.txt
var fh = fileOpen('test.txt', fileOpenRead);
while (!fileEof(fh)) syslog(fileReadLine(fh));
fileClose(fh);
```

10.13 fileReadPos

```
fileReadPos(
    const handle, // file handle
    [const newPos]); // new position if supplied
```

Queries file read pointer. If a new position is given as 2nd argument, the read pointer is moved to that position.

Examples:

```
var pos = fileReadPos(fh); // query current position
fileReadPos(fh, 100); // seek to 100
fileReadPos(fh, fileReadPos(fh)-5); // seek back 5 chars
```

10.14 fileRename

```
fileRename(
    const oldname, // old file name
    const newname) // new file name
```

Renames a file.

Example:

```
fileRename('D:\CONFIG.999', 'D:\CONFIG.888');
```

10.15 fileTempName

```
fileTempName()
```

Creates a unique temporary file name.

Example:

```
var name = fileTempName(); // E:\OS2\IBMCP\TEMP\CslAFRT.C4U
```

10.16 fileWrite

```
fileWrite(
  const handle,    // file handle
  const data)     // data to write
```

Write data to file.

Example:

```
// copy a file
var inp = fileOpen('source.dat',
  fileOpenRead+fileOpenBinary);
var out = fileOpen('target.dat',
  fileOpenWrite+fileOpenBinary);
while (!fileEof(inp))
  fileWrite(out,fileRead(inp,512));
fileClose(inp);
fileClose(out);
```

10.17 fileWriteLine

```
fileWriteLine(
  const handle,    // file handle
  const data)     // data to write
```

Write text line to file.

Example:

```
// copy a file in text mode
var inp = fileOpen('d:\config.sys',fileOpenRead);
var out = fileOpen('d:\config.bak',fileOpenWrite);
while (!fileEof(inp))
  fileWriteLine(out,fileReadLine(inp));
fileClose(inp);
fileClose(out);
```

10.18 fileWritePos

```
fileWritePos(
  const handle,    // file handle
  [const newpos]); // new position if supplied
```

Queries file write pointer. If a new position is given as 2nd argument, the write pointer is moved to that position.

Example:

```
var pos = fileWritePos(fh); // query current position
fileWritePos(fh, 100);     // seek to 100
fileWritePos(fh,fileWritePos(fh)-5); // seek back 5 chars
```

11 Database library

The dax library accesses databases either through their native API (ORACLE, DB2, MySQL) or by ODBC. Prefer the native interface over ODBC for the supported databases, it has less overhead and so is faster.

ODBC has advantages if you don't have a SQL database at all, for example handling DBASE files or connecting to applications like Excel. To access a database where CSL has not yet an native interface you must also use ODBC.

Files

ZcDaxLib.dll / libZcDaxLib.so
The implemented functions

ZcDaxLib.csl
External declarations used when loading at runtime

Globals

Identifier	Description
daxVersion	Current version of the library
daxCompiler	Compiler the library was built by (IBM, BORLAND, MICROSOFT, GNU)
daxBuilt	Date and time when the library was compiled
daxLibtype	Interface the library is using (API or NATIVE = C++ class interface)

11.1 daxCheckCursor

```
daxCheckCursor(  
    const cursor)    // dax cursor handle
```

Check if cursor needs to be (re)parsed.

Returns true if cursor needs to be (re)parsed, false otherwise.

11.2 daxCommit

```
daxCommit(  
    const link)    // db link handle
```

Commit transaction

11.3 daxConnect

```
daxConnect(  
    const database,        // database (ORACLE, DB2, MYSQL)  
    [const connection,    // connection / alias  
    const username,       // user identification  
    const password,       // users password  
    const maxcursor])    // max. cursor pool size (default=16)
```

Connects to a database. Currently *ODBC*, *ORACLE*, *DB2* and *MYSQL* are supported on the platforms they are available for. Returns a database link handle used for subsequent operations.

The meaning of *connection* varies depending on the type of database:

Database	Connection
ODBC	Data source name
Oracle	SQL*Net name
DB2	Database alias
MySQL	database[:host[:port[:socket]]] <u>Examples:</u> test Database <i>test</i> on current host by default port. :boston Default database on host <i>boston</i> . test:50 Database <i>test</i> on current host by port 50

11.4 daxDatabase

```
daxDatabase(
    const link) // db link handle
```

Returns the database type (*ODBC, ORACLE, DB2, MYSQL*)

11.5 daxDisconnect

```
daxDisconnect(
    const link) // database handle
```

Disconnects from database. (Implies a rollback).

11.6 daxDispose

```
daxDispose(
    const cursor) // dax cursor handle
```

Dispose cursor no longer needed so it can be reused for other statements.

Since by default there are max. 16 cursors available you may run out of cursors if you don't dispose cursors no longer needed. (See [daxOpen](#))

11.7 daxDone

```
daxDone(
    const cursor) // db cursor handle
```

Flushes internal write array into the database. Use *daxDone* after *daxParse* and *daxSupply* to complete operation if no other dax-functions are called thereafter.

11.8 daxFetch

```
daxFetch(
    const cursor,    // db cursor handle
    var& vals[],    // target array for one row
    [const nullind]) // null indicators (default = false)
```

Fetch next row from cursor. Returns true if row was fetched, otherwise false (no more data available).

If argument *nullind* is passed as true, each fetched value is made out of 2 variables, where the first is the value itself and the second is a boolean indicating null values by true state. The size of *vals* must therefore always be even.

11.9 daxLiteral

```
daxLiteral(
    const string) // string value
```

Prepares a string as literal input for SQL statements. The string is enclosed in single quotes and single quotes being part of the string value are doubled.

Example:

```
stmt = "insert into test(ident) values('|daxLiteral('joe's place')|)";
// value in stmt:
// insert into test(ident) values('joe''s place')
```

11.10 daxParse

```
daxParse(
    const link,    // db link handle
    const sql,    // SQL statement
    [const maxlong]) // max. select-size for long columns (default 2000)
```

Prepares a SQL statement for subsequent cursor related *dax* functions. Parameters for *daxSupply* are inserted by a # followed by the column size for non-numeric data and a single # for numeric data.

Returns a cursor handle.

11.11 daxRollback

```
daxRollback(
    const link) // db link handle
```

Rolls back the current transaction.

11.12 daxRowsProcessed

```
daxRowsProcessed(
    const cursor) // db cursor handle
```

Returns the number of rows processed

Example:

```
var csr = daxParse(link, "update csctest set descr='xx'");
sysLog(daxRowsProcessed(csr)+' row(s) updated');
```

11.13 daxSelectColumnName

```
daxSelectColumnName(
    const cursor,    // db cursor handle
    const index)    // column index 0-based
```

Returns the name of the column

11.14 daxSelectColumns

```
daxSelectColumns(
    const cursor)    // db cursor handle
```

Returns the number of select columns

11.15 daxSelectColumnSize

```
daxSelectColumnSize(
    const cursor,    // db cursor handle
    const index)    // column index 0-based
```

Returns the size of the column

11.16 daxSelectColumnType

```
daxSelectColumnType(
    const cursor,    // db cursor handle
    const index)    // column index 0-based
```

Returns the type of the column

11.17 daxSimple

```
daxSimple(
    const link,      // db link handle
    const sql)      // sql statement to run
```

Executes a SQL statement. If the statement is a query (SELECT), the first column of the first row is returned as result.

11.18 daxSupply

```
daxSupply(
    const cursor,    // db cursor handle
    const& vals[],  // values to supply
    [const nullind]) // null indicators (default = false)
```

Supplies values for parameter positions in the statement. The number of values supplied in one call is arbitrary, but the total number of values supplied in all calls must correspond to a multiple of the parameters in the statement.

Example:

```
var csr = daxParse(link,
    'insert into csltest (ident, descr) '
    'values (#, #30)');
var x = { 1, 'barbie', 2 };
var y = { 'ken', 3, 'ferrari' };
daxSupply(csr, x);
daxSupply(csr, y);
```

```
daxDone(csr);
...
daxDispose(csr); // if cursor no longer needed
```

If argument *nullind* is passed as true, each supplied value is made out of 2 variables, where the first is the value itself and the second is a boolean indicating null values by true state. The size of *vals* must therefore always be even.

Example:

```
var csr = daxParse(link,
    'insert into csltest (ident, descr) '
    'values (#, #30)'
);
var x = {
    { 1, false }, {'barbie', false },
    { 2, false }, {'', true }
};
daxSupply(csr, x, true);
daxDone(csr);
...
daxDispose(csr); // if cursor no longer needed
```

11.19 Sample 1 (toys.csl)

This is an example using several dax functions. Take a userid, password and connection that is available on your system.

```
#loadLibrary 'ZcSysLib'
#loadLibrary 'ZcStrLib'
#loadLibrary 'ZcDaxLib'

main()
{
    // check arguments
    if (sizeof(mainArgVals) < 3) {
        const exc[3] = {
            'usage : csl toys name/password@connection',
            ' ',
            'example: csl toys SCOTT/TIGER@SALES'
        };
        throw exc;
    }

    syslog('connect');
    var name, pass, conn, a = 2;
    name = strSplitConnectString(mainArgVals[a],pass,conn);
    var link = daxConnect('db2',conn,name,pass); // (*) (**)

    try {
        syslog('drop old table');
        daxSimple(link, 'drop table csltest');
        daxCommit(link);
    }
    catch (var exc[]) {
        syslog('no old table to drop');
    }

    daxSimple(link,
        'create table csltest ( '
        'ident integer, ' // (*)
        'descr varchar(30) '
        ') '
    );

    syslog('insert rows');
    var toys = {
        1, 'barbie',
```

```

    12, 'football',
    325, 'tomb raider II',
    18, 'flipper'
};
var csr = daxParse(link,
    'insert into csltest(ident,descr) '
    'values (#, #30)'
);
daxSupply(csr,toys);
daxDone(csr);
daxDispose(csr);
daxCommit(link);

sysLog(
    '# of rows in csltest is '|
    daxSimple(link, 'select count(*) from csltest')
);

sysLog('query rows');
csr = daxParse(link,
    'select ident, descr from csltest '
    'where ident between # and # '
    'order by ident'
);
var vals = { 10, 1000 };
daxSupply(csr, vals);
while (daxFetch(csr, vals))
    sysLog(vals[0] '|' - '|'vals[1]);

sysLog('disconnect');
daxDisconnect(link);
}

```

For ORACLE you would have to change the statements marked (*) to:

```

var link = daxConnect('oracle',conn,name,pass);
...
daxSimple(link,
    'create table csltest ( '
    'ident number(6), '
    'descr varchar2(30) '
    ') '
);

```

For change to MYSQL you would only have to change line (**).

If you want to write database independent dax scripts have a closer look to the Sample 2 (Portable).

11.20 Sample 2 (portable.csl)

This is a modified version of the example 1 (toys) showing how to write scripts that run on either database without modifications.

```

#loadLibrary 'ZcSysLib'
#loadLibrary 'ZcStrLib'
#loadLibrary 'ZcDaxLib'

main()
{
    // check arguments
    if (sizeof(mainArgVals) < 3) {
        const exc[3] = {
            'usage : csl portable db:name/password@connection',
            '      (db defaults to DB2)',
            'example: csl portable SCOTT/TIGER@SALES'
        }
    }
}

```

```

};
throw exc;
}

sysLog('connect');
var db, name, pass, conn, a = 2;
name = strSplitConnectString(mainArgVals[a],pass,conn,db);
if (db == '') db = 'DB2';
var link, integer, varchar;
switch (db) {
  case 'DB2':
  case 'MYSQL':
  case 'ODBC':
    varchar = 'varchar';
    integer = 'integer';
    break;
  case 'ORACLE':
    varchar = 'varchar2';
    integer = 'number(6)';
    break;
  default:
    throw '%% unknown db: '|db;
} // switch
link = daxConnect(db, conn, name, pass);

sysLog('running on '|daxDatabase(link));

try {
  sysLog('drop old table');
  daxSimple(link, 'drop table csltest');
  daxCommit(link);
}
catch (var exc[]) {
  sysLog('no old table to drop');
}

daxSimple(link,
  'create table csltest ( '
  'ident '|integer|', '
  'descr '|varchar|(30)'
  ') '
);

sysLog('insert rows');
var toys = {
  1, 'barbie',
  12, 'football',
  325, 'tomb raider II',
  18, 'flipper'
};
var csr = daxParse(link,
  'insert into csltest(ident,descr) '
  'values (#, #30)'
);
daxSupply(csr,toys);
daxDone(csr);
daxDispose(csr);
daxCommit(link);

sysLog(
  '# of rows in csltest is '|
  daxSimple(link, 'select count(*) from csltest')
);

sysLog('query rows');
csr = daxParse(link,
  'select ident, descr from csltest '
  'where ident between # and # '

```

```

    'order by ident'
);
var vals = { 10, 1000 };
daxSupply(csr, vals);
while (daxFetch(csr, vals))
    syslog(vals[0]|' - '|vals[1]);

sysLog('disconnect');
daxDisconnect(link);
}

```

11.21 Sample 3 (unknown.csl)

Example selecting an table with unknown column layout

```

#loadLibrary 'ZcSysLib'
#loadLibrary 'ZcStrLib'
#loadLibrary 'ZcDaxLib'

main()
{
    // check arguments
    if (sizeof(mainArgVals) < 4) {
        const exc[3] = {
            'usage : csl unknown [db:]name/password@connection tablename',
            '',
            'example: csl unknown DB2:SCOTT/TIGER@SALES EMP'
        };
        throw exc;
    }

    // connect
    var name, pass, conn, db, a = 2;
    name = strSplitConnectString(mainArgVals[a++],pass,conn,db);
    if (db == '') db = 'MYSQL'; // default db
    var lnk = daxConnect(db,conn,name,pass);

    // start processing
    var csr = daxParse(lnk, 'select * from '|mainArgVals[a]|' order by 1');
    var cols = daxSelectColumns(csr);
    var line, size[cols];

    // display title
    for (var i=0; i<cols; i++) {
        size[i] = daxSelectColumnSize(csr,i);
        line |=
            strSubString(
                daxSelectColumnName(csr,i),
                1, size[i]+1
            );
    }
    syslog(line);

    // underline title
    line = '';
    for (i=0; i<cols; i++)
        line |= strSubString(' ',1,size[i],'-')|' ';
    syslog(line);

    // query and display the rows
    var col[cols];
    while (daxFetch(csr, col)) {
        line = '';
        for (i=0; i<cols; i++)
            line |= strSubString(col[i],1,size[i])|' ';
        syslog(line);
    }
}

```

```
sysLog();  
sysLog(daxRowsProcessed(csr) | ' row(s) selected');  
daxDisconnect(lnk);  
}
```

The output might look like this:

```
IDENT  DESCR  
-----  
1      leather football  
7      tennis racket  
15     rollerblades  
  
3 row(s) selected
```


12 Async Communication

Files

ZcComLib.dll / libZcComLib.so
The implemented functions

ZcComLib.csl
External declarations used when loading at runtime

Globals

Identifier	Description
comVersion	Current version of the library
comCompiler	Compiler the library was built by (IBM, BORLAND, MICROSOFT, GNU)
comBuilt	Date and time when the library was compiled
comLibtype	Interface the library is using (API or NATIVE = C++ class interface)

12.1 comBits

```
comBits(  
    const handle,    // async handle  
    [const bits])   // # of bits per char
```

Query and/or set the number of bits per character. If parameter *bits* is omitted, the current setting is queried. Valid settings are 5, 6, 7 and 8.

Examples:

```
var bits = comBits(hdl); // query current setting  
comBits(hdl, 7); // set # of bits per char
```

12.2 comBps

```
comBps(  
    const handle,    // async handle  
    [const bps])    // bits per second
```

Query and/or set the speed of the communication line. If parameter *bps* is omitted, the current setting is queried. Valid settings are 300, 600, 1200, 2400 and so on.

Examples:

```
var bps = comBps(hdl); // query current setting  
comBps(hdl, 38400); // set speed
```

12.3 comClose

```
comClose(
    const handle)    // async handle
```

Close handle and release communication line.

Example:

```
var hdl = comOpen('com1','com1.log');
comBps(hdl,9600);
comWrite(hdl, 'hello world');
comWaitForOutput(hdl);
comClose(hdl);
```

12.4 comFlow

```
comFlow(
    const handle,      // async handle
    [const flow])     // flow control mode: N, H, S, B
```

Query and/or set the flow control mode. If parameter *flow* is omitted, the current setting is queried. Valid modes:

Mode	Flow Control
N	No flow control
H	Hardware flow control (sent: CTS, receive: DTR)
S	Software flow control XON/XOFF (DC1/DC3)
B	Both: hardware and software control

Examples:

```
var flow = comFlow(hdl); // query current setting
comFlow(hdl, 'h'); // enable hardware flow control
```

12.5 comInputChars

```
comInputChars(
    const handle)    // async handle
```

Query # of characters in input queue. Note there may be in fact more characters waiting in the device input buffer than reported by `comInputChars`. Use this function only to find out if any character will be returned by the next read or not. Don't use this function to wait until a certain amount of characters are available, that may not work on all platforms.

Example:

```
var n = comInputChars(hdl);
if (n > 0)
    syslog('Next read will return at least '|n|' chars');
else
    syslog('Next read will wait');
```

12.6 comOpen

```
comOpen(
    const devname,    // name of async device
    [const logfile]) // optional communication logfile
```

Opens a communication line and returns a handle. The handle returned by *comOpen* is needed in subsequent calls to other COM functions.

devname is usually something like *com1*, *com2* etc. Please note that Win32 requires a special notation for device names other than *com1...com9*. You will have to write `\\.\` in front of all non-standard device names. For example you might have to write `\\.\com10` instead of *com10*.

Devices are always initialized to these settings by *comOpen*:

- 9600 Bits per second
- 8 databits per word
- no parity
- 1 stopbit
- no flow control
- read timeout 1000 milliseconds

These settings may be altered by calling *comBps*, *comBits*, *comParity*, *comStops* and *comReadTimeout* after opening the device.

Optionally all communications can be traced into a *logfile*. The logfile is plain ASCII and may be inspected by any text editor.

Example:

```
var hdl = comOpen('com1','com1.log');
comBps(hdl,9600);
comWrite(hdl, 'hello world');
comWaitForOutput(hdl);
comClose(hdl);
```

12.7 comParity

```
comParity(
    const handle,    // async handle
    [const parity]) // parity code: N, E, O, M, S
```

Query and/or set the parity mode. If parameter *parity* is omitted, the current setting is queried. Valid parity code settings:

Code	Parity
N	None
E	Even
O	Odd
M	Mark (always 1)
S	Space (always 0)

Examples:

```
var par = comParity(hdl); // query current setting
comParity(hdl, 'e'); // set parity even
```

12.8 comPurgeInput

```
comPurgeInput(
  const handle) // async handle
```

Purges all input waiting for readout by *comRead* or *comReadChar*.

Example:

```
comPurgeInput(hdl);
```

12.9 comRead

```
comRead(
  const handle, // async handle
  [const maxchars]) // # of bits per char
```

Reads up to *maxchars* characters from the input queue. *comRead* returns immediately when at least 1 character is read. If no character is available, *comRead* will wait for a character up to the read timeout value.

If *maxchars* is omitted, 1 is assumed.

Example:

```
var hdl = comOpen('com1');
comReadTimeout(hdl, 5000);
comWrite(hdl, 'Send a character, you have 5 seconds: ');
var data = comRead(hdl, 10);
if (data == '')
  comWrite(hdl, 'timed out');
else
  comWrite(hdl, 'thanks!');
comClose(hdl);
```

12.10 comReadChar

```
comReadChar(
  const handle) // async handle
```

Reads a character from the input queue. If no character is available, *comReadChar* will wait for a character up to the current read timeout setting.

comReadChar does a buffered read minimizing system overhead. It performs better for single character processing than *comRead*.

Example:

```
var hdl = comOpen('com1');
comReadTimeout(hdl, 5000);
comWrite(hdl, 'Send a character, you have 5 seconds: ');
var data = comReadChar(hdl);
if (data == '')
  comWrite(hdl, 'timed out');
else
  comWrite(hdl, 'thanks!');
comClose(hdl);
```

12.11 comReadTimeout

```
comReadTimeout(
  const handle,      // async handle
  [const millisecs]) // new timeout value
```

Query and/or set the read timeout in milliseconds. If parameter *millisecs* is omitted, the current setting is queried. A timeout value of 0 will wait *forever*.

Examples:

```
var rto = comReadTimeout(hdl); // query current setting
comReadTimeout(hdl, 30000); // set timeout to 30 seconds
```

12.12 comStops

```
comStops(
  const handle,      // async handle
  [const stopbits]) // # of stopbits
```

Query and/or set the number of stopbits per character. If parameter *stopbits* is omitted, the current setting is queried. Valid settings are 1, 1.5, and 2.

Examples:

```
var stops = comStops(hdl); // query current setting
comStops(hdl, 2); // set # of stopbits
```

12.13 comWaitForOutput

```
comWaitForOutput(
  const handle) // async handle
```

Wait until all pending output chars have been processed and sent.

Example:

```
comWaitForOutput(hdl);
```

12.14 comWrite

```
comWrite(
  const handle, // async handle
  const data)  // data to send
```

Send *data* to the receiver.

Note that processing is usually done asynchronously: *comWrite* puts the data into a sending queue and returns immediately. If you have to be sure that your output has completed before your program proceeds, use *comWaitForOutput*.

Examples:

```
comWrite(hdl, 'hello world\n');
```


13 Registry/Profile handling

WARNING

Using functions of this library can seriously damage your system if you are not sure about what you are doing or if your script has errors. You are therefore urgently advised to make backups of your registry or system profiles respectively while developing and testing scripts with this library.

This library is available for Windows and OS/2 only.

Files

ZcPrfLib.dll

The implemented functions

ZcPrfLib.csl

External declarations used when loading at runtime

Globals

Identifier	Description
<code>prfVersion</code>	Current version of the library
<code>prfCompiler</code>	Compiler the library was built by (IBM, BORLAND, MICROSOFT, GNU)
<code>prfBuilt</code>	Date and time when the library was compiled
<code>prfLibtype</code>	Interface the library is using (API or NATIVE = C++ class interface)
<code>prfTypeString</code> <code>prfTypeBinary</code> <code>prfTypeInteger</code> <code>prfTypeOther</code> <code>prfTypeAuto</code>	Constants for <i>prfSetValue</i> and <i>prfValueType</i>

13.1 prfClose

```
prfClose(           // close profile
  const handle);   // profile handle
```

Closes a profile handle not longer used.

13.2 prfDeleteKey

```
prfDeleteKey(      // delete key in path
  const handle,    // profile handle
  const key);      // key
```

Delete a key and all it's values and sub keys.

OS/2: The path must be empty, there are no keys below a path.

13.3 prfDeleteValue

```
prfDeleteValue(      // delete value
    const handle,    // profile handle
    [const name]);   // value name
```

Delete a value from current path.

Windows: Omitting the value name will delete the default value of the key.

OS/2: The value name is mandatory.

13.4 prfGetKeys

```
var prfGetKeys(      // get list of keys in path
    const handle,    // profile handle
    [var& keys[]]); // array for key names (def=query count only)
```

Get a list of the keys in the current path. Returns number of keys found.

OS/2: The path must be empty, there are no keys below a path.

13.5 prfGetValue

```
var prfGetValue(     // read value
    const handle,    // profile handle
    [const name]);   // value name (def=''=standard value)
```

Read value.

Windows: Omitting the name will read the standard value.

OS/2: Both, path and value name must not be empty.

13.6 prfGetValues

```
var prfGetValues(    // get list of values in path
    const handle,    // profile handle
    [varvalues[]]); // array for value names (def=query count only)
```

Read list of value names in current path. Returns number of values found.

OS/2: Path must not be empty.

13.7 prfKeyExists

```
var prfKeyExists(    // query key in path existence
    const handle,    // profile handle
    [const key]);    // key (def=''=query path existence)
```

Check if key exists. Omitting the key will query if the current path exists.

13.8 prfOpen

```
var prfOpen(           // open a profile
    const root,       // root name
    [const path]);    // path name
```

Opens registry or profile respectively and returns a handle for subsequent calls to prf functions.

Windows: if root does not start with one of

- HKEY_LOCAL_MACHINE
- HKEY_CURRENT_USER
- HKEY_CLASSES_ROOT
- HKEY_USERS
- HKEY_PERFORMANCE_DATA
- HKEY_CURRENT_CONFIG
- HKEY_DYN_DATA

the root name will be prefixed by HKEY_LOCAL_MACHINE\Software.

OS/2: Root will be taken as a file name. Usually you will give the file the extension .ini, although any extension may be used.

Path is identical with an application name. Do not use path separators, because OS/2 profiles have only one level of keys (no sub keys).

13.9 prfOpenSystem

```
var prfOpenSystem(    // open system profile
    [const path]);    // path name
```

Opens registry or profile respectively and returns a handle for subsequent calls to prf functions.

Windows: Path will be below HKEY_LOCAL_MACHINE\Software.

OS/2: The system profile is opened.

Path is identical with an application name. Do not use path separators, because OS/2 profiles have only one level of keys (no sub keys).

13.10 prfOpenUser

```
var prfOpenUser(     // open user profile
    [const path]);   // path name
```

Opens registry or profile respectively and returns a handle for subsequent calls to prf functions.

Windows: Path will be below HKEY_CURRENT_USER\Software.

OS/2: The user profile is opened.

Path is identical with an application name. Do not use path separators, because OS/2 profiles have only one level of keys (no sub keys).

13.11 prfPath

```
var prfPath(           // query or set path
    const handle,     // profile handle
    [const path]);    // new path to set
```

Query or change current path. If path is omitted, the current setting is returned.

Windows: Path may be something like aaa\bbb\ccc.

OS/2: Path is identical with an application name. Do not use path separators, because OS/2 profiles have only one level of keys (no sub keys).

13.12 prfSetValue

```
var prfSetValue(      // set value
    const handle,     // profile handle
    const data,       // data to write
    [const name,     // value name (def=''=standard value)
     const type]);    // type enforcement (def=prfTypeAuto)
```

Set a value. If no type is defined it will be determined automatically as:

- If data represents an integer value, the type will be prfTypeInteger.
- If data contains only printable characters type will be set to prfTypeString.
- If both previous conditions are not met, the type will be prfTypeBinary.

Windows: If name is omitted or "", the default value is set.

OS/2: The name is mandatory and a non-empty path must be set.

OS/2 profiles don't really distinguish types, so don't expect prfValueType to return an enforced type correct; instead prfValueType will do it's best to determine the type out of the data itself according to the rules above.

13.13 prfValueExists

```
var prfValueExists(  // query value existence
    const handle,    // profile handle
    [const name]);   // value name (def=''=standard value)
```

Check if value exists.

Windows: If name is omitted or "", the standard value type is queried.

OS/2: The name is mandatory and a non-empty path must be set.

13.14 prfValueType

```
var prfValueType(    // query value type
    const handle,    // profile handle
    [const name]);   // value name (def=''=standard value)
```

Query a value type.

Windows: If name is omitted or "", the standard value type is queried.

OS/2: The name is mandatory and a non-empty path must be set.

OS/2 profiles don't really distinguish types, so don't expect `prfValueType` to return an enforced type correct; instead `prfValueType` will do it's best to determine the type out of the data itself according to these rules:

- If data represents an integer value, the type will be `prfTypeInteger`.
- If data contains only printable characters type will be set to `prfTypeString`.
- If both previous conditions are not met, the type will be `prfTypeBinary`.

14 Windows control

This library is only available for the Windows platform.

Files

ZcWinLib.dll

The implemented functions

ZcWinLib.csl

External declarations used when loading at runtime

Globals

Identifier	Description
winVersion	Current version of the library
winCompiler	Compiler the library was built by (IBM, BORLAND, MICROSOFT, GNU)
winBuilt	Date and time when the library was compiled
winLibtype	Interface the library is using (API or NATIVE = C++ class interface)
VK_MENU VK_LMENU VK_RMENU	Code of ALT keys for winPostVKey(). (ALT key is used to call menu)
VK_F1 ... VK_F24	Function key codes for winPostVKey()
VK_SHIFT VK_LSHIFT VK_RSHIFT	Code of SHIFT keys for winPostVKey().
VK_CONTROL VK_LCONTROL VK_RCONTROL	Code of CTRL keys for winPostVKey()
VK_NUMPAD0 ... VK_NUMPAD9	Code of keys on numeric keypad for winPostVKey().
VK_LEFT VK_UP VK_RIGHT VK_DOWN	Cursor key codes for winPostVKey()
VK_LBUTTON VK_RBUTTON VK_MBUTTON VK_CANCEL VK_BACK VK_TAB VK_CLEAR VK_RETURN VK_PAUSE VK_CAPITAL VK_ESCAPE VK_SPACE VK_PRIOR VK_NEXT VK_END VK_HOME VK_SELECT VK_PRINT VK_EXECUTE VK_SNAPSHOT VK_INSERT VK_DELETE VK_HELP VK_LWIN VK_RWIN VK_APPS VK_MULTIPLY VK_ADD VK_SEPARATOR VK_SUBTRACT VK_DECIMAL VK_DIVIDE VK_NUMLOCK VK_SCROLL VK_PROCESSKEY VK_ATTN VK_CRSEL VK_EXSEL VK_EREOF VK_PLAY	Other virtual key codes for winPostVKey(). Refer to windows documentation if usage or purpose of these virtual keys need to be explained.

```
VK_ZOOM VK_NONAME
VK_PA1 VK_OEM_CLEAR
```

14.1 winActivate

```
winActivate(
  const handle); // windows handle
```

Brings the window to the foreground. Window handles may be found by function [winFind](#)

14.2 winClose

```
winClose(
  const handle); // windows handle
```

Closes the window. Window handles may be found by function [winFind](#)

14.3 winCapsLock

```
var winCapsLock(
  [const mode]); // change mode
```

Query or change state of the CAPS LOCK key depending on *mode*:

Mode	Action
(none)	Query current state of CAPS LOCK
0	Turn CAPS LOCK off
1	Turn CAPS LOCK on
2	Toggle CAPS LOCK state

Return value:

```
true = CAPS LOCK is now on
false = CAPS LOCK is now off
```

14.4 winFind

```
var winFind(
  const name, // name or regular expression pattern
  var [], // handle name of each match
  [const regular]); // use regular expression (default = false)
```

Find windows by the title text. *name* is either the exact title or a [regular expression](#). The results are stored as pairs of window handles and titles in the array *win*.

Returns the number of windows found. Note that the number of windows found may be larger than the *win* array size.

14.5 winHide

```
winHide(
  const handle); // windows handle
```

Hides the window. The window may be shown again with [winShow](#). Window handles may be found by function [winFind](#).

14.6 winIsMaximized

```
var winIsMaximized(
  const handle); // windows handle
```

Query if the window is maximized. Window handles may be found by function [winFind](#).

14.7 winIsMinimized

```
var winIsMinimized(
  const handle); // windows handle
```

Query if the window is minimized. Window handles may be found by function [winFind](#).

14.8 winIsVisible

```
var winIsVisible(
  const handle); // windows handle
```

Query if the window is visible. Visible is a rather technical term, because a *visible* window may be covered completely by others. Window handles may be found by function [winFind](#).

14.9 winMaximize

```
winMaximize(
  const handle); // windows handle
```

Maximize the window. Maximizing a window can be reversed by [winRestore](#). Window handles may be found by function [winFind](#).

14.10 winMinimize

```
winMinimize(
  const handle); // windows handle
```

Minimize the window. Minimizing a window can be reversed by [winRestore](#). Window handles may be found by function [winFind](#).

14.11 winNumLock

```
var winNumLock(
    [const mode]); // change mode
```

Query or change state of the NUM LOCK key depending on *mode*:

Mode	Action
(none)	Query current state of NUM LOCK
0	Turn NUM LOCK off
1	Turn NUM LOCK on
2	Toggle NUM LOCK state

Return value:

true = NUM LOCK is now on
false = NUM LOCK is now off

14.12 winPostText

```
winPostText(
    const text); // text
```

Post text to the active window. A window may be activated by [winActivate](#). Window handles may be found by function [winFind](#).

14.13 winPostVKey

```
winPostVKey(
    const vkey1, // virtual key code VK_...
    [const vkey2, // virtual key code VK_...
     const vkey3]); // virtual key code VK_...
```

Post a virtual key or a combination of up to 3 virtual keys to the active window. A window may be activated by [winActivate](#).

In addition to the [predefined virtual keys](#) found in *ZcWinLib.csl*, the ASCII codes of uppercase letters A...Z and digits 0...9 may be used as virtual keys, as *strAscii('A')* for example.

Example:

```
winPostVKey(VK_MENU, strAscii('F')); // post Alt-F
```

14.14 winPrintScreen

```
var winPrintScreen(
    [const mode]); // region selection
```

Simulate a press of the PRINTSCREEN key in order to obtain a screen snapshot and save it to the clipboard. The region may be selected by *mode*:

Mode	Region
(none) or 0	All screen

1 Active window

Return value:

true = NUM LOCK is now on
false = NUM LOCK is now off

14.15 winRestore

```
winRestore(
    const handle); // windows handle
```

Restore a minimized or maximized window to its original position and size. Window handles may be found by function [winFind](#)

14.16 winScrollLock

```
var winScrollLock(
    [const mode]); // change mode
```

Query or change state of the SCROLL LOCK key depending on *mode*:

Mode	Action
(none)	Query current state of SCROLL LOCK
0	Turn SCROLL LOCK off
1	Turn SCROLL LOCK on
2	Toggle SCROLL LOCK state

Return value:

true = SCROLL LOCK is now on
false = SCROLL LOCK is now off

14.17 winShow

```
winShow(
    const handle); // windows handle
```

Make window visible if it was previously hidden (for example by [winHide](#)). Although visible the window may still be invisible because others are covering it. You may in case use [winActivate](#) to bring the window to the top. Window handles may be found by function [winFind](#)

14.18 Sample (notepd.csl)

This sample uses several functions of the window control library.

It launches notepad.exe and finds the corresponding window handle. Then some text is written to the client area and the file is saved.

Finally there are minimizing/maximizing operations run and notepad is closed again.

NOTE:

Depending on the version and language of your notepad.exe you will have to change the title text in the winFind()

function below to run the sample. Otherwise the notepad window may not be found.

```
#loadLibrary 'ZcSysLib'
#loadLibrary 'ZcWinLib'
#loadLibrary 'ZcFileLb'

/*
 * w i n W a i t F o r
 *
 * Wait until a window with given title is open
 */
static var winWaitFor(const title, const maxsecs)
{
    const start = sysElapsed();
    var win[10][2]; // space for window records
    while (!winFind(title, win)) {
        sysSleep(300); // wait a while
        if (sysElapsed()-start >= maxsecs)
            throw '%% window '|title|' not found!';
    } // for
    return win[0][0]; // handle
} // winWaitFor

main()
{
    // Launch notepad.exe. We use the "start" command since we want
    // sysCommand to return immediately and not to wait till notepad
    // has ended:
    syslog('starting notepad...');
    sysCommand('start notepad.exe');

    // Now wait for notepad to start.
    // I used a german notepad.exe of Windows 2000, for other locales
    // or versions the title must be changed accordingly:
    syslog('waiting for window...');
    const handle = winWaitFor('Unbenannt - Editor', 5);

    // make sure window is activated
    syslog('activate window...');
    winActivate(handle);

    // write text to the client area
    syslog('post some text...');
    winPostText('Hello world!');

    syslog('save file...');
    // make sure file does not exist in advance:
    try { fileDelete('C:\\CslTest.txt'); } catch (var exc[]) {}

    // select 'save as..' by the menu:
    winPostVKey(VK_MENU); // open menu
    winPostVKey(VK_DOWN); // move down to "save as..."
    winPostVKey(VK_DOWN);
    winPostVKey(VK_DOWN);
    winPostVKey(VK_DOWN);
    winPostText('\n'); // could also use winPostVKey(VK_RETURN) here

    // write file name into file dialog and save by enter:
    winPostText('C:\\CslTest.txt\n');

    // Now fool around with the window
    syslog('maximize...');
    winMaximize(handle);
    sysSleep(1000);

    syslog('restore...');
    winRestore(handle);
    sysSleep(1000);
}
```

```
sysLog('minimize...');
winMinimize(handle);
sysSleep(1000);

sysLog('restore...');
winRestore(handle);
sysSleep(1000);

// Close notepad. This could of cause also be done with VKeys.
// Using winClose is a 'hard' close giving the application no
// chance to complain anything.
sysLog('close window...');
winClose(handle);

sysPrompt('press enter to finish...');
} // main
```


15 C API

The C *application programming interface* (API) enables to embed CSL in your own application as a macro language or write your own libraries for CSL.

The C API uses the standard linkage of the system (`__stdcall` for Windows, `_System` for OS/2) so in fact every compiler should be able to use it. However for compilers not listed below there are usually some modifications necessary in the header file `ZBase.h`. In case the object format differs from microsoft, a new LIB file must be created by a tool like `IMPLIB`.

Below you find the list of currently tested compilers. If you have adapted and tested another successfully, please let [me](#) know.

Compiler	Version(s)	Sample dir
IBM VisualAge C++ for OS/2	3.0.8	.\Samples\API\Os2\Ibm
IBM VisualAge C++ for Windows	3.5.9	.\Samples\API\Win\Ibm
Microsoft Visual C++	5.0	.\Samples\API\Win\Microsoft
Borland C++ (free commandline tools)	5.5	.\Samples\API\Win\Borland
GNU GCC and mingw32	2.95.2	.\Samples\API\Win\Mingw32
GNU GCC and cygwin	2.91.57 / B20.1	.\Samples\API\Win\Cygwin
GNU GCC on unixish systems	2.95.2	./Samples/API/Unix/Gnu

15.1 Embedding CSL

Embedding CSL in your application is quite easy: After opening a CSL handle the API functions can be used to compile scripts, define and call functions, query and set variables and so on.

Have a look at this example making use of various API functions. You will find it as *Embed.c* in the `Samples\API\Source` directory:

```
#include <stdlib.h>
#include <stdio.h>
#include <ZCslApi.h>

static char* module = "Embed"; /* module name */
static ZCslHandle csl; /* csl handle */
static long errs; /* csl api return code */

/*
 * c h e c k E r r s
 *
 * This function checks for errors. Error messages are in case displayed
 * and program is terminated to keep the sample as simple as possible.
 */
static void checkErrs(void)
{
    long size, errs2;
    int i;
    char *msg;

    if (errs) {
        fprintf(stderr, "ZCsl error found:\r\n");
        for (i = 0; i < errs; i++) {
            /* First we get the size of the message only so a buffer */

```

```

/* large enough for the string can be allocated: */
errs2 = ZCslGetError(csl, i, 0, ; /* get size of message */
if (errs2) { errs = errs2; checkErrs(); }

/* Now a buffer is allocated and the message is retrieved */
/* and displayed: */
msg = (char*)malloc(size);
errs2 = ZCslGetError(csl, i, msg, ; /* get message */
if (errs2) { errs = errs2; checkErrs(); }
fprintf(stderr, "%s\r\n", msg);

/* the buffer is no longer required and gets discarded: */
free(msg);
} /* for */
exit(1);
} /* if */
} /* checkErrs */

/*
 * c h e c k N u m b e r
 *
 * Check if string represents a number
 */
static int checkNumber(char *s)
{
    int any;

    any = 0;
    if (*s=='-' || *s=='+') s++;
    while ('0'<=*s *s<='9') { s++; any = 1; }
    if (*s=='.') s++;
    while ('0'<=*s *s<='9') s++;
    return any *s == 0;
} /* checkNumber */

/*
 * a v e r a g e
 *
 * Sample CSL function calculating the average of up 5 numbers
 */
ZExportAPI(void) average(ZCslHandle aCsl)
{
    int argCount, i;
    long bufsiz;
    double sum;
    char buf[40], name[4];

    /* get actual # of arguments */
    bufsiz = sizeof(buf);
    ZCslGet(aCsl, "argCount", buf, ;
    argCount = atoi(buf);

    /* calculate sum of all arguments */
    sum = 0.0;
    for (i = 0; i < argCount; i++) {
        /* create name of parameter */
        sprintf(name, "p%d", i+1);

        /* get argument */
        bufsiz = sizeof(buf);
        if ( ZCslGet(aCsl, name, buf, ) return; /* (1) */

        /* check for number */
        if (!checkNumber(buf)) {
            sprintf(buf, "argument p%d is no number!", i+1);
            ZCslSetError(aCsl, buf, -1); /* (2) */
        } /* if */
    }
}

```

```

    sum += atof(buf);
} /* for */

/* return result */
sprintf(buf, "%f", sum / argCount);
ZCslSetResult(aCsl, buf, -1); /* (2) */
} /* average */

int main()
{
    long bufsize;
    char buf[1024];
    static char *vals[] = { "3", "12", "17" };

    printf("opening csl\r\n");
    errs = ZCslOpen(0);
    checkErrs();

    printf("\r\nget csl version\r\n");
    bufsize = sizeof(buf);
    errs = ZCslGet(csl, "cslVersion", buf, ; /* (1) */
    checkErrs();
    printf("  csl version is %s\r\n", buf);

    printf("\r\nload C function 'average'\r\n");
    errs = ZCslAddFunc(
        csl,                                /* handle */
        module,                             /* module name */
        "average(const p1, "                /* declaration */
            "[const p2, "
            "const p3, "
            "const p4, "
            "const p5])",
        average);                           /* function address */
    checkErrs();

    printf("\r\ncall 'average' from C\r\n");
    errs = ZCslCall(
        csl,                                /* handle */
        module,                             /* module name */
        "average",                          /* function to call */
        sizeof(vals)/sizeof(char*),        /* # of arguments */
        vals);                               /* arguments */
    checkErrs();

    printf("\r\nget result\r\n");
    bufsize = sizeof(buf);
    errs = ZCslGetResult(csl, buf, ; /* (1) */
    checkErrs();
    printf("  result = %s\r\n", buf);

    printf("\r\ncompile a script from memory\r\n");
    errs = ZCslLoadScriptMem(
        csl,                                /* handle */
        module,                             /* module name */

        "#loadLibrary 'ZcSysLib'\n"        /* the script */
        ""
        "check()\n"
        "{\n"
        "  syslog('the average of 3, 5, 12 and 7 is '\n"
        "    |average(3,5,12,7)\n"
        "  );\n"
        "}\n"
    );
    checkErrs();

    printf("\r\ncall 'check' within compiled script\r\n");

```

```

errs = ZCslCall(csl, module, "check", 0, 0);
checkErrs();

printf("\r\nclosing csl\r\n");
errs = ZCslClose(csl);
checkErrs();

return 0;
} /* main */

```

Notes:

1. Within implementations of CSL functions there is no need to handle errors since that will be done by the caller. Just return in case of an error.
2. These functions will always return a value of 0, so there is no need to check for errors.

15.2 Writing libraries

Writing a CSL library is easy and gives you the opportunity to provide a professional script interface to your application.

You should make yourself familiar with the concept of DLL's or shared libraries in your compiler documentation. For your convenience you can consult the files *build.bat* / *build.cmd* / *build* in the *Samples\API* subdirectories to see what compiler and linker switches are required.

Your library must export 2 entries: *ZCslInitLib* and *ZCslCleanupLib*.

ZCslInitLib is called when the DLL gets loaded. You use the API to define global var's and const's and load functions at startup. *ZCslCleanupLib* will be called when the DLL is unloaded so you can perform any tidy up before the CSL handle is closed.

You will find this sample library *MyLib.c* in the *Samples\API\Source* directory. Use the sample as a starting point for your own DLL's:

```

#include <stdlib.h>
#include <stdio.h>
#include <ZCslApi.h>

/*
 * c h e c k N u m b e r
 *
 * Check if string represents a number
 */
static int checkNumber(char *s)
{
    int any;
    any = 0;
    if (*s=='-' || *s=='+') s++;
    while ('0'<=*s *s<='9') { s++; any = 1; }
    if (*s=='.') s++;
    while ('0'<=*s *s<='9') s++;
    return any *s == 0;
} /* checkNumber */

/*
 * a v e r a g e
 *
 * Sample CSL function calculating the average of up 5 numbers
 */
ZExportAPI(void) average(ZCslHandle aCsl)
{
    double sum;
    long bufsiz;

```

```

int argCount, i;
char buf[40], name[4];

/* get actual # of arguments */
bufsiz = sizeof(buf);
ZCslGet(aCsl, "argCount", buf, ;
argCount = atoi(buf);

/* calculate sum of all arguments */
sum = 0.0;
for (i = 0; i < argCount; i++) {
    /* create name of parameter */
    sprintf(name, "p%d", i+1);

    /* get argument */
    bufsiz = sizeof(buf);
    if ( ZCslGet(aCsl, name, buf, ) return; /* (1) */

    /* check for number */
    if (!checkNumber(buf)) {
        sprintf(buf, "argument %d is no number!", i);
        ZCslSetError(aCsl, buf, -1); /* (2) */
    } /* if */

    sum += atof(buf);
} /* for */

/* return result */
sprintf(buf, "%f", sum / argCount);
ZCslSetResult(aCsl, buf, -1); /* (2) */
} /* average */

/*
 * i n i t i a l i z e
 *
 * initialize csl library at load time
 */
ZCslInitLib(csl)
{
    static char* module = "MyLib"; /* module name */
    long errs;

    /* define a global constant by loading a script */
    errs = ZCslLoadScriptMem(
        csl, /* csl handle */
        module, /* module name */
        "const myVersion = 1.0;\n" /* script source */
    );
    if (errs) return; /* (1) */

    /* load a function */
    ZCslAddFunc(
        csl, /* handle */
        module, /* module name */
        "average(const p1, " /* declaration */
            "const p2, "
            "[const p3, "
            "const p4, "
            "const p5])",
        average); /* function address */
    /* no errs check since returning anyway */
} /* initialize */

/*
 * c l e a n u p
 *
 * clean up csl library before unloading
 */

```

```
ZCslCleanupLib(csl)
{
    /* nothing to clean up in our sample */
} /* cleanup */
```

15.3 API reference

This section lists all API functions in alphabetic order.

Each API function returns a long value indicating how many error texts have been emitted. This count can be used to fetch all error texts with *ZCslGetError*. If the function was completed with no errors, 0 is returned.

15.3.1 ZCslAddFunc

```
long ZCslAddFunc(                /* add a function */
    ZCslHandle aHandle,          /* CSL handle */
    const char *aFileName,       /* file/module name for function */
    const char *aFuncHeader,     /* function header */
    void (ZFuncptrAPI aFunc)(ZCslHandle aHandle) /* function address */
);
```

Adds a C/C++ function to CSL.

15.3.2 ZCslAddVar

```
long ZCslAddVar(                /* add a local var/const */
    ZCslHandle aHandle,          /* CSL handle */
    const char *aVarName,        /* name and layout of var/const */
    const char *aInitVal,        /* initial value */
    long aIsConst                /* 0=var, 1=const */
);
```

Adds a local var or const within a C/C++ function implementation.

aInitVal may be *NULL* if var/const ought to be initialized to an empty string.

NOTE: Can *NOT* be used to add globals. To add a global var or const you may compile a script with it's declaration.

15.3.3 ZCslCall

```
long ZCslCall(                  /* call function */
    ZCslHandle aHandle,          /* CSL handle */
    const char *aFileName,       /* file/module caller belongs to */
    const char *aFuncName,       /* function name */
    long aArgCount,              /* # of arguments following */
    char *aParam[]               /* parameter list. NULL if no args */
);
```

Calls any CSL, C or C++ function known to CSL. Parameters are passed as a list of ASCIZ strings.

Example:

```
static char *args[] = { "The quick brow fox", "5", "10" };
long errs, size;
char buf[40];

/* call strSubString */
errs = ZCslCall(csl, "MyProg.exe", "strSubString", sizeof(args)/sizeof(char*), args);
if (errs) .... /* error handling */
```

```

/* retrieve result */
size = sizeof(buf);
errs = ZCslGetResult(csl, buf, &size);
if (errs) .... /* error handling */

```

15.3.4 ZCslCallEx

```

long ZCslCallEx(          /* call function (extended) */
    ZCslHandle aHandle,   /* CSL handle */
    const char *aFileName, /* file/module caller belongs to */
    const char *aFuncName, /* function name */
    long aArgCount,       /* # of arguments following */
    char *aParam[],       /* parameter list. NULL if no args */
    long aSize[]          /* parameter size list. NULL if no args */
);

```

Calls any CSL, C or C++ function known to CSL. Parameters are supplied as pairs of char* and long. The long value is the size of the corresponding argument. If it is -1 the parameter is considered a zero terminated string. With this API it is possible to pass parameters containing NUL chars.

Example:

```

static char *args[] = { "The quick brow fox", "5", "\33[c\0\1" };
static long szel[] = { -1, -1, 5 };
long errs, size;
char buf[40];

/* call strSubString */
errs = ZCslCallEx(csl, "MyProg.exe", "strSubString", sizeof(args)/sizeof(char*), args, szel);
if (errs) .... /* error handling */

/* retrieve result */
size = sizeof(buf);
errs = ZCslGetResult(csl, buf, &size);
if (errs) .... /* error handling */

```

15.3.5 ZCslClose

```

long ZCslClose(          /* Destroy CSL Context & release mem */
    ZCslHandle aHandle   /* CSL handle */
);

```

Closes a CSL handle. All DLL's are unloaded and allocated memory is released.

15.3.6 ZCslGet

```

long ZCslGet(           /* get var or const value */
    ZCslHandle aHandle, /* CSL handle */
    const char *aVarName, /* variable name */
    char *aBuffer,       /* buffer for value (NULL = query size) */
    long *aSize          /* buffer size */
);

```

Get value of any CSL variable or constant.

On entry *aSize* must be set to the actual size of the buffer. No more than this amount will be filled including zero termination. The rest of the variable content will in case be truncated. On exit *aSize* is always set to the buffer size required for holding the full value.

Pass *aBuffer* as NULL to only query the required buffer size.

If *aSize* is NULL, the API assumes your buffer is in any case large enough and will not limit the size at all (I strongly don't recommend this!).

Example:

```
long size, errs;
char *buf;

/* get size of value first for buffer allocation: */
errs = ZCslGet(csl, "adr[5][2]", NULL, &size);
if (errs) .... /* error handling */

/* now allocate buffer and retrieve value */
buf = (char*)malloc(size);
errs = ZCslGet(csl, "adr[5][2]", buf, &size);
if (errs) .... /* error handling */

...

/* when value no longer needed, don't forget to release buffer */
free(buf);
```

15.3.7 ZCslGetError

```
long ZCslGetError(          /* get error text */
    ZCslHandle aHandle,    /* CSL handle */
    long aIndex,           /* index of text (0 based) */
    char *aBuffer,        /* buffer for error text (NULL = query size) */
    long *aSize            /* buffer size information */
);
```

Retrieve error text. *aIndex* must be in the range from 0 to the # of errors returned by the previous API call.

On entry *aSize* must be set to the actual size of the buffer. No more than this amount will be filled including zero termination. The rest of the variable content will in case be truncated. On exit *aSize* is always set to the buffer size required for holding the full value.

Pass *aBuffer* as NULL to only query the required buffer size.

If *aSize* is NULL, the API assumes your buffer is in any case large enough and will not limit the size at all (I strongly don't recommend this!).

Example:

```
void handleCslError(ZCslHandle csl, long errs)
{
    long i, size, errs2;
    char *buf;

    for (i = 0; i < errs; i++) {
        /* get size of text first for buffer allocation: */
        errs2 = ZCslGetError(csl, i, NULL, &size);
        if (errs2) handleCslError(csl, errs2);

        /* now allocate buffer and retrieve text */
        buf = (char*)malloc(size);
        errs2 = ZCslGetError(csl, i, buf, &size);
        if (errs2) handleCslError(csl, errs2);

        /* display message */
        fprintf(stderr, "%s\r\n", buf);

        /* when value no longer needed, don't forget to release buffer */
        free(buf);
    }
}
```

```

    } /* for */
    if (errs) exit(1);
} /* handleCslError */

```

15.3.8 ZCslGetResult

```

long ZCslGetResult(          /* get function result */
    ZCslHandle aHandle,      /* CSL handle */
    char *aBuffer,           /* buffer for value (NULL = query size) */
    long *aSize              /* buffer size */
);

```

Retrieve return value of a CSL function call.

On entry *aSize* must be set to the actual size of the buffer. No more than this amount will be filled including zero termination. The rest of the variable content will in case be truncated. On exit *aSize* is always set to the buffer size required for holding the full value.

Pass *aBuffer* as NULL to only query the required buffer size.

If *aSize* is NULL, the API assumes your buffer is in any case large enough and will not limit the size at all (I strongly don't recommend this!).

Example:

```

static char *args[] = { "The quick brow fox", "5", "10" };
char *buf;
long errs, size;

/* call strSubString */
errs = ZCslCall(csl, "MyProg.exe", "strSubString", sizeof(args)/sizeof(char*), args);
if (errs) .... /* error handling */

/* get size of return value first for buffer allocation: */
errs = ZCslGetResult(csl, NULL, &size);
if (errs) .... /* error handling */

/* now allocate buffer and retrieve value */
buf = (char*)malloc(size);
errs = ZCslGetResult(csl, buf, &size);
if (errs) .... /* error handling */

...

/* when value no longer needed, don't forget to release buffer */
free(buf);

```

15.3.9 ZCslLoadLibrary

```

long ZCslLoadLibrary(        /* load a CSL dll */
    ZCslHandle aHandle,      /* CSL handle */
    const char *aDllName     /* dll filename */
);

```

Loads a dynamic linked library and calls *ZCslInitLib*. If the file name is given without extension, .dll or .so.X is assumed.

NOTE: CSL will not load the same library several times; attempt to do so will be silently ignored.

15.3.10 ZCslLoadScriptFile

```
long ZCslLoadScriptFile( /* load script from file */
    ZCslHandle aHandle, /* CSL handle */
    const char *aFileName /* file name to load script */
);
```

Compiles/loads a script from a file. If the file name is given without extension, *.csl* is appended. The file is searched in the directories listed in environment variable *CSLPATH*. If *CSLPATH* does not exist, the file is searched in the current working directory.

NOTE: CSL will not load the same file/module several times; attempt to do so will be silently ignored.

15.3.11 ZCslLoadScriptMem

```
long ZCslLoadScriptMem( /* load script from memory */
    ZCslHandle aHandle, /* CSL handle */
    const char *aFileName, /* file/module name for script */
    const char *aStr /* csl source code */
);
```

Compiles/loads a script from memory.

NOTE: CSL will not load the same file/module several times; attempt to do so will be silently ignored.

15.3.12 ZCslOpen

```
long ZCslOpen( /* Create CSL context & alloc mem */
    ZCslHandle *aHandle, /* address of handle var */
    long aFlags /* initialization flags */
);
```

aFlags is a combination of the following values (bitwise or them together):

Flag	Description
ZCslDisInclude	Disable #include
ZCslDisLoadLibrary	Disable #loadLibrary
ZCslDisLoadScript	Disable #loadScript
ZCslDisList	Disable #list
ZCslDisLogFile	Disable #logFile
ZCslDisIf	Disable #if / #else / #endif
ZCslDisError	Disable #error
ZCslDisMessage	Disable #message
ZCslDisAll	Disable all compiler directives

Set *aFlags* to 0 to enable all directives.

15.3.13 ZCslSet

```
long ZCslSet(                                /* set var or const value */
    ZCslHandle aHandle,                      /* CSL handle */
    const char *aVarName,                    /* variable name */
    const char *aValue,                      /* new value (NULL = "") */
    long aSize                                /* value size, -1 = ASCIZ */
);
```

Set value of any CSL global or local variable.

Example:

```
long errs;
errs = ZCslSet(csl, "adr[5][2]", "Fred Flintstone", -1);
if (errs) .... /* error handling */
```

15.3.14 ZCslSetError

```
long ZCslSetError(                          /* set/add error text */
    ZCslHandle aHandle,                      /* CSL handle */
    const char *aBuffer,                    /* buffer for result */
    long aSize                               /* buffer size, -1 = ASCIZ */
);
```

Set/add an error text. This function is used in C/C++ function implementations to raise an error. *ZCslSetError* may be called multiple to add several texts.

ZCslSetError will always return 0 so there is no need for error checking.

Example:

```
ZExportAPI(void) mthSqrt(ZCslHandle aCsl)
{
    char buf[40];
    long bufsiz;
    double val;

    /* get val */
    bufsiz = sizeof(buf);
    if ( ZCslGet(aCsl, "val", buf, &bufsiz) ) return;
    val = atof(buf);

    if (val < 0.0) {
        ZCslSetError(aCsl, "val must not be negative!", -1);
        return;
    } /* if */

    /* return result */
    sprintf(buf, "%f", sqrt(val));
    ZCslSetResult(aCsl, buf, -1);
} /* mthSqrt */
```

15.3.15 ZCslSetResult

```
long ZCslSetResult(          /* set return result */
    ZCslHandle aHandle,     /* CSL handle */
    const char *aBuffer,    /* buffer for result */
    long aSize              /* buffer size, -1 = ASCIZ */
);
```

Set return value in a C/C++ function implementation.

ZCslSetResult will always return 0 so there is no need for error checking.

Example:

```
ZExportAPI(void) mthSqrt(ZCslHandle aCsl)
{
    char buf[40];
    long bufsiz;
    double val;

    /* get val */
    bufsiz = sizeof(buf);
    if ( ZCslGet(aCsl, "val", buf, &bufsiz) ) return;
    val = atof(buf);

    if (val < 0.0) {
        ZCslSetError(aCsl, "val must not be negative!", -1);
        return;
    } /* if */

    /* return result */
    sprintf(buf, "%f", sqrt(val));
    ZCslSetResult(aCsl, buf, -1);
} /* mthSqrt */
```

15.3.16 ZCslSetTraceMode

```
long ZCslSetTraceMode(      /* set trace mode */
    ZCslHandle aHandle,     /* CSL handle */
    long aMode              /* trace mode to set */
);
```

Set [trace](#) mode. Predefined mode constants are:

Mode	Description
ZCslTraceNone	Trace output is turned off (default)
ZCslTraceCode	Trace P-code instructions together with the 2 top stack elements.
ZCslTraceFuncs	Trace function entry and exit. Trace output will be indented within the function.
ZCslTraceBlks	Trace entry and exit of named blocks. Trace output will be indented within the block. Use #block to name blocks.
ZCslTraceMsgs	Trace expressions of trace statement or API ZCslTrace and ZCsl::trace member.

The mode constants above may be combined by or-ing them. Combined constants are predefined as:

Mode	Description
ZCslTraceInfo	ZCslTraceFuncs ZCslTraceBlks ZCslTraceMsgs

15.3.17 ZCslShow

```
long ZCslShow(          /* show internal informations */
    ZCslHandle aHandle, /* CSL handle */
    long aMode,         /* show mode */
    long aDepth         /* how much, -1 = all */
);
```

Show internal information of CSL. Requires system library loaded.

Available information (mode's):

Mode	Description
ZCslShowFunctions	Show all known functions
ZCslShowCallStack	Show call path with all functions.
ZCslShowFullStack	Show call path with all functions an all local variables
ZCslShowGlobals	Show globals and static identifiers of all files/modules
ZCslShowLibraries	Show all libraries loaded

15.3.18 ZCslStartDateTime

```
long ZCslStartDateTime(          /* get start date and time */
    ZCslHandle aHandle,         /* CSL handle */
    long *aYear,                 /* year (NULL = not required) */
    long *aMonth,                /* month 1...12 (NULL = not required) */
    long *aDay,                  /* day 1...31 (NULL = not required) */
    long *aHour,                 /* hour 0...23 (NULL = not required) */
    long *aMinute,               /* minute 0...59 (NULL = not required) */
    long *aSecond                /* second 0...59 (NULL = not required) */
);
```

Get CSL start date (date and time of *ZCslOpen* call). For values you don't need you may pass a NULL pointer.

Example:

```
long errs, year, month, day, hour, minute;
errs =
    ZCslStartDateTime(
        csl,
        &year, &month, &day);
    &hour, &minute, NULL);
if (errs) .... /* error handling */
printf(
    "CSL was started at %02ld/%02ld%/%04ld %02ld:%02ld%\r\n",
    month, day, year, hour, minute);
```

15.3.19 ZCslTrace

```
long ZCslTrace(          /* trace a message */
    ZCslHandle aHandle, /* CSL handle */
    const char *aMessage /* message to display when tracing */
);
```

Provide trace information. The message will only be displayed when tracing messages is enabled.

Example:

```
long errs;
errs = ZCslTrace(csl, "my message");
if (errs) .... /* error handling */
```

15.3.20 ZCslTraceMode

```
long ZCslTraceMode(      /* query trace state */
    ZCslHandle aHandle, /* CSL handle */
    long *aMode          /* returned mode value */
);
```

Get current trace mode. (See [ZCslSetTraceMode](#) for mode values)

Example:

```
long errs, trace;
errs = ZCslTraceMode(csl, &trace);
if (errs) .... /* error handling */
if (trace != ZCslTraceNone)
    puts("trace is on\r\n");
else
    puts("trace is off\r\n");
```

15.3.21 ZCslVarExists

```
long ZCslVarExists(      /* check var/const existence */
    ZCslHandle aHandle, /* CSL handle */
    const char *aVarName /* name var/const */
);
```

Checks if the var/const was declared. Does not check if it also is implemented or if any given indexes are valid. (Works like *exists* operator at compile time). Returns 0/1.

15.3.22 ZCslVarResize

```
long ZCslVarResize(      /* resize variable */
    ZCslHandle aHandle, /* CSL handle */
    const char *aVarName /* name and new layout of variable */
);
```

Resize a local array (like *resize* operator)

Example:

```
/* resize list to hold 20 entries */
long errs = ZCslVarReize(csl, "list[20]");
if (errs) .... /* error handling */
```

15.3.23 ZCslVarSizeof

```

long ZCslVarSizeof(          /* get size of variable */
    ZCslHandle aHandle,     /* CSL handle */
    const char *aVarName,   /* name of variable */
    long *aSize             /* buffer for size */
);

```

Get size of any local variable or constant (like *sizeof* operator)

Example:

```

/* how big is list? */
long errs, size;
errs = ZCslVarSizeof(csl, "list", &size);
if (errs) .... /* error handling */

```


16 C++ Class Interface

The C++ class interface is based on the IBK ZClass library which is part of the CSL package and can be used together with most other class libraries without conflicts. It is currently available for following configurations:

Compiler	Version(s)	Sample dir
IBM VisualAge C++ for OS/2	3.0.8	.\Samples\Class\Os2\Ibm
IBM VisualAge C++ for Windows	3.5.9	.\Samples\Class\Win\Ibm
Microsoft Visual C++	5.0	.\Samples\Class\Win\Microsoft
Borland C++ (free commandline tools)	5.5	.\Samples\Class\Win\Borland
GNU GCC on unixish systems	2.95.2	./Samples/Class/Unix/Gnu

The binary distributions for Windows and OS/2 include only the class library for IBM VisualAge C++. If you have a Borland or Microsoft compiler you must build the class library first. See Readme.txt in `.\Win\Borland` or `.\Win\Microsoft` respectively for instructions.

Although the class library works fine with GCC on Linux, this is not the case for 2.95.2 on Windows. It seems exceptions cannot be handled across dll's: Throwing an exception in a dll and not catching it in the same dll will make GNU terminate the program with SIGABORT instead of passing the exception to the calling dll or exe.

In fact there are 2 implementations of the CSL class interface:

- If you include `ZCsl.hpp` you are working directly with the CSL kernel. I call this the NATIVE interface. It requires however that the CSL kernel and your application are built with the same compiler. This is the directest interface to CSL.
- If you include `ZCslWrap.hpp` instead, every call will be redirected to the C API, which in turn will call the native interface. This allows mixed compiler configurations. There is an additional overhead by this interface, but in practice you'll hardly recognize a difference. The libraries of the binary windows distribution are built with this interface.

I'm sorry but the documentation of the ZClass library is still in my queue. Don't know when I will find time to do it (any volunteers?). You may help yourself by consulting the header files and source.

16.1 Embedding CSL

Embedding CSL in your application is quite easy. All you have to do is create an instance of `ZCsl`. Then you may use member functions to manipulate CSL. When processing is done, the `ZCsl` object can be deleted.

Have a look at the example below using various methods of `ZCsl`. You will find this as `Embed.csl` in the `.\Samples\Source` directory):

```
#include <stdio.h>
#include <ZExcept.hpp>

#if ZC_WIN
    #include <ZCslWrap.hpp>
#else
    #include <ZCsl.hpp>
#endif

#if ZC_GNU
#include <strstream.h>
```

```

#else
#include <strstream.h>
#endif

static ZCsl* csl = 0;           // csl object ptr
static ZBoolean cslOk(zFalse); // ok flag
static ZString module("Embed"); // module name

/*
 * c h e c k N u m b e r
 *
 * Check if string represents a number
 */
static ZBoolean checkNumber(const ZStringstr)
{
    const char *s = str;
    if (*s=='-' || *s=='+') s++;
    ZBoolean any(zFalse);
    while ('0'<=*s *s<='9') { s++; any = zTrue; }
    if (*s=='.') s++;
    while ('0'<=*s *s<='9') s++;
    return any *s == 0;
} // checkNumber

/*
 * a v e r a g e
 *
 * Sample CSL function calculating the average of numbers
 */
static ZString average(ZCsl* aCsl)
{
    // get actual # of arguments
    int argCount = aCsl->get("argCount").asInt();

    // calculate sum of all arguments
    double sum(0.0);
    for (int i = 0; i < argCount; i++) {
        // get argument
        ZString val = aCsl->get("p"+ZString(i+1));

        // check for number
        if (!checkNumber(val))
            ZTHROWEXC("argument "+ZString(i+1)+" is no number!");

        sum += val.asDouble();
    } // for

    // return result
    return ZString(sum / argCount);
} // average

int main()
{
    int ret(0);
    try {
        cout << "creating csl object" << endl;
        csl = new ZCsl();
        cslOk = zTrue;

        cout << endl << "get csl version" << endl;
        cout << "  csl version is " << csl->get("cslVersion") << endl;

        cout << endl << "load c++ function 'average'" << endl;
        csl->addFunc(
            module,
            "average(const p1, [const p2, const p3, const p4, const p5])",
            average);
    }
}

```

```

cout << endl << "call 'average' from c++" << endl;
ZString res = csl->call(module, "average", 3, "3", "12", "17");
cout << "  result = " << res << endl;

cout << endl << "compile a script from memory" << endl;
istream str(
    "#loadLibrary 'ZcSysLib'\n"
    ""
    "check()\n"
    "{\n"
    "  syslog('the average of 3,5,12,7 is '\n"
    "    |average(3,5,12,7)\n"
    "  );\n"
    "}\n"
);
csl->loadScript(module, ;

cout << endl << "call 'check' within compiled script" << endl;
csl->call(module, "check");

cout << endl << "deleting csl object" << endl;
delete csl;
} // try
catch (const ZExceptionerr) {
    for (int i = 0; i < err.count(); i++)
        cerr << err[i] << endl;
    if (cslOk) delete csl;
    ret = 1;
} // catch
return ret;
} // main

```

Instead of loading the script from memory as in the example above you could also load the script from a file of course:

```

...
csl.loadScript("User.csl");
...

```

16.2 Writing libraries

Writing a CSL library is easy and gives you the opportunity to provide a professional script interface to your application.

You should make yourself familiar with the concept of DLL's or shared libraries in your compiler documentation. For your convenience you can consult the files *build.bat* / *build.cmd* / *build* in the *Samples\Class* subdirectories to see what compiler and linker switches are required.

Your library must export 2 entries: *ZCslInitLib* and *ZCslCleanupLib*.

ZCslInitLib is called when the DLL gets loaded. You use the API to define global var's and const's and load functions at startup. *ZCslCleanupLib* will be called when the DLL is unloaded so you can perform any tidy up before the CSL handle is closed.

You will find this sample library *MyLib.cpp* in the *Samples\Class\Source* directory. Use the sample as a starting point for your own DLL's:

```

#include <ZBase.h> // load ZC_.... defines

#if ZC_GNU
    #include <strstream.h>
#else
    #include <strstrea.h>
#endif

#if ZC_WIN

```

```

#include <ZCslWrap.hpp>
#else
#include <ZCsl.hpp>
#endif

static ZString myStrStrip(ZCsl* csl)
{
    return csl->get("string").strip();
} // myStrStrip

static ZString mySubString(ZCsl* csl)
{
    int argc = csl->get("argCount").asInt();
    switch (argc) {
        case 2:
            return csl->get("string").subString(
                csl->get("start").asInt()
            );
        case 3:
            return csl->get("string").subString(
                csl->get("start").asInt(),
                csl->get("count").asInt()
            );
        default:
            return csl->get("string").subString(
                csl->get("start").asInt(),
                csl->get("count").asInt(),
                csl->get("padchar")[1]
            );
    } // switch
} // mySubString

ZCslInitLib(csl)
{
    ZString iFile("MyLib");
    istrstream init("const myVersion = 0.1;\n");
    csl->loadScript(iFile, ;
    (*csl)
        .addFunc(
            iFile,
            "myStrStrip(const string)",
            myStrStrip)
        .addFunc(
            iFile,
            "mySubString(const string, const start, [const count, const padchar])",
            mySubString);
} // ZCslInitLib

ZCslCleanupLib(aCsl)
{
    // nothing to cleanup in this sample
} // ZCslCleanupLib

```

16.3 Class interface reference

This section lists all members of ZCsl.

16.3.1 ZCsl constructor

```
ZCsl::ZCsl(           // constructor
    int aFlags=0)    // compiler directive flags
```

The flags argument is a combination of the bit values:

Flag	Description
ZCsl::disInclude	Disable #include
ZCsl::disLoadLibrary	Disable #loadLibrary
ZCsl::disLoadScript	Disable #loadScript
ZCsl::disList	Disable #list
ZCsl::disLogFile	Disable #logFile
ZCsl::disIf	Disable #if / #else / #endif
ZCsl::disError	Disable #error
ZCsl::disMessage	Disable #message
ZCsl::disAll	Disable all compiler directives

By default all directives are enabled

16.3.2 ZCsl destructor

```
ZCsl::~ZCsl(); // destructor
```

The destructor will first call the cleanup-functions of all loaded libraries, then unload the libraries from memory and finally drop all other memory occupied.

16.3.3 ZCsl::addFunc

```
ZCsl& ZCsl::addFunc(           // add a C++ function
    const ZString& aFileName,   // file/module belonging to
    const ZString& aFuncHeader, // function header
    ZString (*aFunc)(ZCsl* aCsl)); // C++ function address
```

Adds a C++ function to CSL

16.3.4 ZCsl::addVar

```
ZCsl& ZCsl::addVar(           // add a local var/const
    const ZString& aVarName,   // name and layout of var/const
    const ZString& aInitVal="", // initial value
    ZBoolean aIsConst=zFalse   // false=var, true=const
```

Adds a local var or const within a C++ function implementation.

NOTE: Can *NOT* be used to add globals. To add a global var or const you may compile a script with it's declaration.

16.3.5 ZCsl::call

Overload 1

```
ZString ZCsl::call(           // call function
    const ZString& aFileName, // file/module caller belongs to
    const ZString& aFuncName, // function name
    long aArgCount = 0,       // # of arguments
    char** aArgs = 0,         // argument list
    long* aSize = 0);         // argument length's (-1 = asciz)
```

Calls any function known to CSL. Arguments are passed as array of char pointers, and optionally an array of long's with the length of each argument. aSize may be NULL if all args are ASCIZ. Single args of type ASCIZ may be indicated by a size of -1.

Overload 2

```
ZString ZCsl::call(           // call a function
    const ZString& aFileName, // file/module name of caller
    const ZString& aFuncName, // function name
    int aArgCount,           // # of arguments
    ...);                    // arguments of type char*
```

Calls any function known to CSL. All arguments are assumed to be ASCIZ.

16.3.6 ZCsl::callEx

```
ZString ZCsl::callEx(        // call a function (extended)
    const ZString& aFileName, // file/module name of caller
    const ZString& aFuncName, // function name
    int argCount = 0,         // # of arguments
    ...);                     // argument pairs of type char* / long
```

Calls any function known to CSL. Arguments are supplied as pairs of char* and long. The long value is the length of the preceding argument. If it is -1 the argument is considered a zero terminated string (ASCIZ).

16.3.7 ZCsl::get

```
ZString ZCsl::get(           // get a var/const value
    const ZString& aVarName); // var or const name
```

Query value of any local, static or global var or const

Examples:

```
double v = csl.get("cslVersion").asDouble();
```

```
ZString name = csl.get("address[5][1]");
```

16.3.8 ZCsl::loadLibrary

```
ZCsl& ZCsl::loadLibrary(      // load a library
    const ZString& aDllName); // library name
```

Loads a dynamic linked library and calls *initialize*. If the file name is given without extension, *.dll* is assumed on Windows and OS/2, and *.so.X* is assumed on unixish systems where *X* is the CSL major version.

NOTE: CSL will not load the same library several times; attempt to do so will be silently ignored.

16.3.9 ZCsl::loadScript

Overload 1

```
ZCsl& ZCsl::loadScript(      // load a script from file
    const ZString& aFileName); // file name
```

Compiles/loads a script from a file. If the file name is given without extension, *.csl* is appended. The file is searched in the directories listed in environment variable *CSLPATH*. If *CSLPATH* does not exist, the file is searched in the current working directory.

Overload 2

```
ZCsl& ZCsl::loadScript(      // load a script from istream
    const ZString& aFileName  // file/module name
    istream* aStr);          // input stream
```

Compiles/loads a script from any *istream*. May be used to load a script from memory instead from a file.

NOTE: CSL will not load the same file/module several times; attempt to do so will be silently ignored.

16.3.10 ZCsl::set

```
ZCsl& ZCsl::set(            // set a variable
    const ZString& aVarName, // variable name
    const ZString& aValue=""); // new value
```

Set new value for any local, static or global variable

Examples:

```
double x1(2.5);
int ii(3);
ZString name("pamela");
csl
    .set("xyz", ZString(x1))
    .set("txt", "Hello world")
    .set("addr["+ZString(ii)+"][1]", name);
```

16.3.11 ZCsl::setTraceMode

```
ZCsl& ZCsl::setTraceMode(    // set trace mode
    TraceMode aMode);      // new trace mode
```

Set [trace](#) mode. TraceMode values are:

Mode	Description
ZCsl::traceNone	Trace output is turned off (default)
ZCsl::traceCode	Trace P-code instructions together with the 2 top stack elements.
ZCsl::traceFuncs	Trace function entry and exit. Trace output will be indented within the function.
ZCsl::traceBlks	Trace entry and exit of named blocks. Trace output will be indented within the block. Use #block to name blocks.
ZCsl::traceMsgs	Trace expressions of trace statement or API ZCslTrace and ZCsl::trace member.

TraceMode values above may be combined by or-ing them. Combined values are predefined as:

Mode	Description
ZCsl::traceInfo	ZCsl::traceFuncs ZCsl::traceBlks ZCsl::traceMsgs
ZCsl::traceAll	ZCsl::traceCode ZCsl::traceFuncs ZCsl::traceBlks ZCsl::traceMsgs

16.3.12 ZCsl::show

```
ZCsl& ZCsl::show(          // show CSL internals
    ZCsl::ShowMode aMode, // show what
    long aDepth=99999999); // how much
```

Show internal information of CSL. Requires system library loaded.

Available information (ShowMode's):

Mode	Description
ZCsl::showFunctions	Show all known functions
ZCsl::showCallStack	Show call path with all functions.
ZCsl::showFullStack	Show call path with all functions an all local variables
ZCsl::showGlobals	Show globals and static identifiers of all files/modules
ZCsl::showLibraries	Show all libraries loaded

16.3.13 ZCsl::startDateTime

```
ZDateTime ZCsl::startDateTime(); // get CSL startup date and time
```

Get the CSL startup date and time (time of instance creation)

16.3.14 ZCsl::trace

```
ZCsl& ZCsl::trace( // query tracing mode
    const ZString& aMessage); // message to display when tracing
```

Provide trace information. The message will only be displayed when tracing messages is enabled.

16.3.15 ZCsl::traceMode

```
ZCsl::TraceMode ZCsl::traceMode(); // query tracing mode
```

Queries current trace mode. (See [ZCsl::setTraceMode](#) for TraceMode's)

16.3.16 ZCsl::varExists

```
ZBoolean ZCsl::varExists(
    const ZString& aVarName); // variable name
```

Checks if the var/const was declared. Does not check if it also is implemented or if any given indexes are valid. (Works like *exists* operator at compile time).

16.3.17 ZCsl::varResize

```
ZCsl& ZCsl::varResize(
    const ZString& aVarName); // new variable declaration
```

Changes the size and/or array layout of a variable (like *resize* operator).

16.3.18 ZCsl::varSizeof

```
long ZCsl::varSizeof( // get size of var/const
    const ZString& aVarName); // var or const name
```

Query size of any local, static or global var or const. (Like *sizeof* operator)

17 CSL links

CSL Homepage

<http://csl.sourceforge.net>

Maillist

You may receive news about CSL by joining the maillist on the CSL homepage.

Forums

Support for problems may be obtained by the forums also found on the CSL homepage. You are invited to post your opinions and wishes for future releases, or success stories in these forums. As you are getting experienced in CSL it would be great if you again help novices with your knowledge.

