CA Nimsoft Monitor

C SDK Reference Guide

Version 4.2 Series

Legal Notices

Warranty

The material contained in this document is provided "as is," and is subject to being changed, without notice, in future editions. Further, to the maximum extent permitted by applicable law, Nimsoft LLC disclaims all warranties, either express or implied, with regard to this manual and any information contained herein, including but not limited to the implied warranties of merchantability and fitness for a particular purpose. Nimsoft LLC shall not be liable for errors or for incidental or consequential damages in connection with the furnishing, use, or performance of this document or of any information contained herein. Should Nimsoft LLC and the user have a separate written agreement with warranty terms covering the material in this document that conflict with these terms, the warranty terms in the separate agreement shall control.

Technology Licenses

The hardware and/or software described in this document are furnished under a license and may be used or copied only in accordance with the terms of such license.

No part of this manual may be reproduced in any form or by any means (including electronic storage and retrieval or translation into a foreign language) without prior agreement and written consent from Nimsoft LLC as governed by United States and international copyright laws.

Restricted Rights Legend

If software is for use in the performance of a U.S. Government prime contract or subcontract, Software is delivered and licensed as "Commercial computer software" as defined in DFAR 252.227-7014 (June 1995), or as a "commercial item" as defined in FAR 2.101(a) or as "Restricted computer software" as defined in FAR 52.227-19 (June 1987) or any equivalent agency regulation or contract clause. Use, duplication or disclosure of Software is subject to Nimsoft LLC's standard commercial license terms, and non-DOD Departments and Agencies of the U.S. Government will receive no greater than Restricted Rights as defined in FAR 52.227-19(c)(1-2) (June 1987). U.S. Government users will receive no greater than Limited Rights as defined in FAR 52.227-14 (June 1987) or DFAR 252.227-7015 (b)(2) (November 1995), as applicable in any technical data.

Trademarks

Nimsoft is a trademark of CA.

Adobe®, Acrobat®, Acrobat Reader®, and Acrobat Exchange® are registered trademarks of Adobe Systems Incorporated.

Intel® and Pentium® are U.S. registered trademarks of Intel Corporation.

Java(TM) is a U.S. trademark of Sun Microsystems, Inc.

Microsoft® and Windows® are U.S. registered trademarks of Microsoft Corporation.

Netscape(TM) is a U.S. trademark of Netscape Communications Corporation.

Oracle® is a U.S. registered trademark of Oracle Corporation, Redwood City, California.

UNIX® is a registered trademark of the Open Group.

ITIL® is a Registered Trade Mark of the Office of Government Commerce in the United Kingdom and other countries.

All other trademarks, trade names, service marks and logos referenced herein belong to their respective companies.

Contact CA Support

For your convenience, CA Technologies provides one site where you can access the information that you need for your Home Office, Small Business, and Enterprise CA Technologies products. At http://ca.com/support, you can access the following resources:

- Online and telephone contact information for technical assistance and customer services
- Information about user communities and forums
- Product and documentation downloads
- CA Support policies and guidelines
- Other helpful resources appropriate for your product

Providing Feedback about Product Documentation

Send comments or questions about CA Technologies Nimsoft product documentation to nimsoft.techpubs@ca.com.

To provide feedback about general CA Technologies product documentation, complete our short customer survey which is available on the CA Support website at http://ca.com/docs.

Contents

Chapter 1: The Nimbus API for C

Introduction

The NimBUS API can be grouped into the following groups. The pseudo code below does not show how to use these function calls, rather it shows the sequence in which they are typically used.

General

All NimBUS applications must start and end like this:

```
nimInit(0)
…
nimEnd(0)
```

nimLog

Use NimBUS logging this way:

```
nimLogSet()
nimLog()
```

nimSession

On the client side:

```
nimNamedSession() nimSessionRequest() nimSessionFree()
```

On the server side:

```
nimSessionNewList ()
nimSession(NULL,port) nimSessionAddStdCallback() nimSessionAddCallback()
…
 nimSessionAddList()
loop
     nimSessionDispatch()
end loop

nimSessionFreeList
nimSessionRemoveList
```

nimQoS

Here is a typical sequence when using the nimQoS methods:

```
nimQoSCreate()

nimQoSSendValue()
```

or

```
nimQoSStart()
nimQoSStop()
nimQoSSendTimer()

nimQoSFree()
```

Log Methods

nimLog

Signature

```
void nimLog(int level, char *format, ...);
```

Description
This function will print out an error message with date/time on the format of printf to a logfile.Logfile is set by nimLogSet.

Parameters

int	level	Log level
char *	format	The same format as ***printf***

Returns
None

See also
N/A

nimLogClose

Signature

```
void nimLogClose(void);
```

Description
Closes the log file.

Parameters
N/A

Returns
N/A

See also
N/A

nimLogGetLevel

Signature

```
int nimLogGetLevel();
```

Description
Identify the current log level

Parameters
N/A

Returns
The current log level.

See also
N/A

nimLogInit

Signature

```
int nimLogInit(char *fname,char *prefix);
```

Description
This function will create a log file and set a “prefix” string. Fname can be a filename or the string “stdout”. Use nimLogSetLevel or nimLogSet to change the default log level, or to change the log file. Use nimLog to write to the log file.

Parameters

char *	fname	The name of the log file (can be a file name or the string “stdout”.
char *	prefix	Typically the probe name

Returns
None

See also
N/A

nimLogModule

Signature

```
void nimLogModule (int iLevel,char *pchModule,char *pchFormat,…);
```

Description
Logs messages to the log if the module as been initialized with nimLogSetModule. Logging is to the same file as set up by calls to nimLogSet.

Parameters

int	iLevel	Log level for this message
char *	pchModule	For module name logging

char *	*pchFormat	The same format as printf

See also
N/A

nimLogPDS

Signature

```
void nimLogPds(PDS *pds,char *print_name,int log_level,int flags);
```

Description
Write a PDS to the log.

Parameters

PDS	pds	PDS to write to the log file
char *	print_name	Log header
int	log_level	Log level
int	flags	Detail level

Returns
N/A

See also
nimLog

nimLogSet

Signature

```
int nimLogSet(char *fname,char *prefix,int level,int flags);
```

Description
This function will create a log file and set a “pre-” string and log level. Fname can be a file name or the string “stdout”. Ue nimLog to write to the logfile.

Parameters

char *	fname	The name of the log file (can be a file name or the string “stdout”.)
char *	prefix	Typically the probe name
int	level	Log level
int	flags	
NIM_LOGF_NOTRUNCATE	flags	Do not truncate the log
NIM_LOGF_RESETATSTART	flags	Truncate the log file at the start of
NIM_LOGF_THREAD_ID	flags	Print the thread ID as part of the log.message.

Returns
NIME_OK or NIME_INVAL

See also
N/A

nimLogSetLevel

Signature

```
int nimLogSetLevel(int level);
```

Description
Change the current log level

Parameters

int	level	Set new log level

Returns
Old log level

See also
N/A

nimLogSetModule

Signature

```
void nimLogSetModule (char *pchModule)
```

Description
Initialize logging for specific modules

Parameters

char * pchModule List of modules e.g. “A,B,C”

See also
N/A

nimLogSession

Signature

```
void nimLogSession (NIMSESS *nims, char *file)
```

Description
Log information about the supplied session object.

Parameters

NIMSESS *nims		Session object to log information for
char *file		File name to log to; overwriting if it exists

See Also
N/A

nimLogTruncate

Signature

```
void nimLogTruncate(void);
```

Description
Truncate current log file now

Parameters
N/A

Returns
N/A

nimLogTruncateSize

Signature

```
void nimLogTruncateSize (unsigned log bytes);
```

Description
Truncate the log based on size bytes instead of the default 100KB. Overridden by time-based truncation if that has been set.

Parameters

unsigned	long bytes	Log file can be made before it is truncated

See Also
nimLogTruncateTime

nimLogTruncateTime

Signature

```
void nimLogTruncateTime(time_t seconds);
```

Description
Truncate log based on age of file (in seconds) instead of file size. Will override size-based truncation.

Parameters

time_t	seconds	Age of file from probe startup or last truncation until the log should be truncated, regardless of size.

See Also
nimLogTruncateSize

Communication Methods

nimAlarm

Signature

```
int nimAlarm(char *message, int pri, char *suppression, char *subsys, char *source,
char **nimid);
```

Description
Used to send a NimBUS alarm. If you need to send more than one alarm within a short period of time use the nimSessionAlarm function instead.

Parameters

char *	message	Alarm message
int	pri	Severity (0-5) E.g. NIML_CRITICAL
char *	suppression	Suppression key
char *	subsys	Subsystem
char *	source	Set source, NULL = host name
char **	nimid	NimBUS identifier for the alarm

Returns
NIME_OK or an error code.

See also
nimSessionAlarm

nimCallbackReply

Signature

```
extern int  nimCallbackReply(NIMCB *cb, int status , PDS *dpds);
```

Description
Must be called in all session callbacks in order to send a reply to the caller of the command (nimRequest). The caller will receive the status and data that is set in the callback function. The data PDS must be freed in the callback before the function ends. Earlier versions of the API did not have this function; instead operated in this way: nimSendReply(cb->msg, status, dpds);

Parameters

NIMCB *	cb	Callback handle
int	status	Command status
PDS *	dpds	Command result data

Returns
NIME_OK on success or an error code.

See also
nimSendReply

nimChangePassword

Signature

```
int  nimChangePassword(char *name, char *old_password, char *new_password);
```

Description
Change the password for the given user. Old password must be correct in order to change the password.

Parameters

char *	name	User name
char *	old_password	Old password
char *	new_password	New password

Returns
NIME_OK or an error code.

See also
N/A

nimChangeLogin

Signature

```
int nimChangeLogin(char *sid);
```

Description
Can be used to switch between two or more different logins.

Parameters

char	*sid*	A valid SID

Returns
NIME_OK or an error code

See also
nimLogin

nimGetCurrentSid

Signature

```
char  *nimGetCurrentSid();
```

Description
Returns a pointer to the current SID in use by the NimBUS API. NB! Do not modify this value.

Parameters
N/A

Returns
Pointer to the current SID

See also
nimChangeLogin

nimEnd

Signature

```
int   nimEnd(int flag);
```

Description
Release all initialized NimBUS data. This should be the last NimBUS call in any application.

Parameters

int	*flag*	Should be 0

Returns
NIME_OK

See also
nimInit

nimError2Txt

Signature

```
const char  *nimError2Txt(int errorcode);
```

Description
This function maps the nimbus errorcode to constant string. If errorcode is out of range, it will return the string "unknown error".

Parameters

int	*errorcode*	NimBUS error code

Returns
OK: The error string
Error: None

See also
N/A

nimInit

Signature

```
int   nimInit(int flag);
```

Description
Initialize the NimBUS API. On windows that includes the WinSock initializing. This should be the first NimBUS call in any application.

Parameters

int	*flag*	Should be 0

Returns
NIME_OK

See also
nimEnd

nimLogin

Signature

```
int   nimLogin(char *name, char *password, char **login_sid);
```

Description
Login to NimBUS with user name and password. The login is global and will affect all subsequently calls to NimBUS. You must free the SID when done with it.

Parameters

char *	name	NimBUS user name
char *	password	Password
char **	login_sid	The Session Identifier (SID)

Returns
NIME_OK or an error code.

See also
nimLogout

nimLogout

Signature

```
void  nimLogout();
```

Description
Log off the current user. All global user credentials will be removed.

Parameters
N/A

Returns
N/A

See also
nimLogin

nimMsgFree

Signature

```
void  nimMsgFree(NIMMSG *nmsg);
```

Description
For internal use; releases a message handle.

Parameters

NIMMSG *	*nmsg*	Pointer to a NimBUS message handle.

Returns
N/A

See also
nimSessionGetMsg

nimNamedRequest

Signature

```
int   nimNamedRequest(char *address, char *cmd, PDS *data, PDS **retData, int
secWait);
```

Description
Send a NimBUS request to the probe address and wait for a reply.

Parameters

char *	address	NimBUS address
char *	cmd	NimBUS command
PDS *	data	Parameter PDS for the request
PDS **	retData	Reply PDS
int	secWait	Timeout

Returns
NIME_OK or an error code.

See also
nimNamedSession

nimNamedSession

Signature

```
NIMSESS *nimNamedSession(char *address);
```

Description
Create a session connected to the given NimBUS address. nimSessionRequest is used to issue requests to the probe when the connection is up.

Parameters

char *	*address*	NimBUS address to which to connect the session

Returns
NIMSESS or NULL on failure.

See also
nimSessionRequest, nimSessionFree

nimPostMessage

Signature

```
int   nimPostMessage(char *subject, int pri, char *suppression, PDS *data, char
**nimid);
```

Description

Post a message with a given subject to NimBUS. This is the generic method that is the basis for both nimAlarm and nimQoS messages.

Parameters

char *	subject	Message subject
int	pri	Set to 0
char *	suppression	Suppression key, normally NULL
PDS *	data	Data to post
char **	nimid	NimBUS identifier for the message

Returns
NIME_OK or an error code.

See also
N/A

nimRegisterProbe

Signature

```
int   nimRegisterProbe(char *probename, int port);
```

Description
This function will register the active session with nimbus name (probename/port) at the controller. You can get the portnumber for the active session by acessing nims->iPort.

Parameters

char *	probename	The probe name
int	port	Port to register

Returns
NIME_OK or an error code.

See also
nimUnRegisterProbe.

nimRequest

Signature

```
int   nimRequest(char *addr,int iPort, char *cmd, PDS *data, PDS **retdata, int secWait);
```

Description
Send a NimBUS request to the probe address and wait for reply. This function is often used together with nimGetNameToIp. We recommend that you use nimNamedRequest instead.

Parameters

char *	addr	IP or host name to connect to
int	iPort	Port number to connect to
char *	cmd	NimBUS command
PDS *	data	Paramter PDS for the request
PDS **	retData	Reply PDS
int	secWait	Time out

Returns
NIME_OK or an error code.

See also
nimNamedRequest

nimSendReply

Signature

```
int   nimSendReply(NIMMSG *nmsg, int status , PDS *dpds);
```

Description
For internal use; used to reply to a nimRequest.

Parameters

NIMMSG *	nmsg	Pointer to a NimBUS message
int	status	Status
PDS *	dpds	Data

Returns
NIME_OK on success or an error code.

See also
N/A

nimSession

Signature

```
NIMSESS     *nimSession (char *host, int port);
```

Description
Create a NimBUS session for client or server. This convenience function will create a server session on “port” if the host parameter is set to NULL. If a hostname is passed on then a create client session is attempted.

Parameters

char *	host	Host name
int	port	Port number

Returns
NIMSESS * on success
NULL if an error occurs.

See also
nimSessionNew, nimSessionServer, nimSessionConnect.

nimSessionAttachCallback

Signature

```
void nimSessionAttachCallback (NIMSESS *nims, NIMCB *cb)
```

Description
Attach a callback object to the session specified.

Parameters

NIMSESS *	nims	Session to attach callback object to.
NIMCB *	cb	Callback object to attach to session.

See also nimSessionDetachCallback

nimSessionAddCallbackPds

Signature

```
int  nimSessionAddCallbackPds NIMSESS *nims,char *cmd, int (*func)(NIMCB *cb,PDS
*pdsArgs), char *fmt, int sec_rights)
```

Description
This function replaces nimSessionAddCallback, and is the required function to use on 64-bit systems, which pass integer parameters. This is due to the way the callback dispatcher works, in that it cannot mix 32-bit and 64-bit parameters in the function call. This mechanism works around that by passing all parameters in a PDS, which the user is responsible to unpacking in the callback. The keys and types are listed in the format string. The function returns NIME_OK on success or an error code on failure.

Parameters

NIMSESS *	nims	Session to add command to
char *	cmd	Command name
int(*func)	(NIMCB*cb,PDS*pdsArgs)	Function pointer to callback function which takes a NIMCB pointer and a PDS pointer as parameters and returns int
char *	fmt	Callback's parameter format string
int	sec_rights	Security level required to execute this callback command

Note
The function will exit with a no-restart code if you attempt to use it with integer parameters on 64-bit systems.

See also
nimSessionAddCallback, nimSessionNew, nimSessionServer

nimSessionDetachCallback

Signature

```
NIMCB *nimSessionDetachCallback(NIMSESS *nims, char *cmd)
```

Description
Remove a callback command from the session.

Parameters

NIMSESS *	nims	Session to detach command from
char *	cmd	Command to detach.

See also
nimSessionAttachCallback

nimSessionNew

Signature

```
NIMSESS     *nimSessionNew();
```

Description
This function will create a new NimBUS session. Default type is stream.(TCP). Use nimSessionSetDatagram change type.

Parameters
N/A

Returns
Pointer to session (NIMSESS) on success.
NULL if an error occurred.

See also
nimSessionFree, nimSessionSetDatagram.

nimSessionRequest

Signature

```
int   nimSessionRequest(NIMSESS *nims, char *cmd, PDS *data, PDS **retData, int
secWait);
```

Description
Send a request to a probe and wait for the answer. Request parameters and the reply are in the form of a PDS.

Parameters

NIMSESS *	nims	Pointer to a session
char *	cmd	Requested command
PDS *	data	Input to the command
PDS **	retData	Data returned from the command
int	secWait	Timeout

Returns
NIME_OK or an error code.

See also
nimRequest

nimSessionFree

Signature

```
void  nimSessionFree(NIMSESS *nims);
```

Description
This function will free resources allocated to this session and free itself.

Parameters

NIMSESS *	*nims*	NimBUS session handle

Returns
None.

See also
nimSessionNew

nimSessionGraceClose

Signature

```
Void nimSessionGraceClose (NIMSESS *nims, int nsec)
```

Description
Shut down a session and free it.

Parameters

NIMSESS *	nims	Session to close
int	nsec	At present not in use.

See Also
N/A

nimSessionNewList

Signature

```
NIMSESSLIST *nimSessionNewList(void);
```

Description
Create a new NimBUS session list. Session lists are typically used by server applications (probes) to handle multiple client sessions (nimSessionDispatch).

Parameters
N/A

Returns
NIMSESSLIST pointer or NULL on failure.

See also
nimSessionDispatch

nimSessionAddList

Signature

```
void  nimSessionAddList(NIMSESSLIST *listhead, NIMSESS *nims);
```

Description
Add a NimBUS session to a session list.

Parameters

NIMSESSLIST *	listhead	Pointer to a session list
NIMSESS *	nims	Pointer to a session.

Returns
N/A

See also
nimSessionNewList, nimSessionRemoveList

nimSessionRemoveList

Signature

```
void  nimSessionRemoveList(NIMSESSLIST *listhead, NIMSESS *nims);
```

Description
Remove a session from a session list. Does not free the NIMSESS so you must call nimSessionFree for the session after it has been removed from the list.

Parameters

NIMSESSLIST *	listhead	Pointer to a session list
NIMSESS *	nims	Pointer to a session

Returns
N/A

See also
nimSessionAddList, nimSessionFree

nimSessionFreeList

Signature

```
void  nimSessionFreeList(NIMSESSLIST **listhead);
```

Description
Free a session list.

Parameters

NIMSESSLIST ** *listhead* A Session list

NIMSESS *	nims	Session to change

Returns
N/A

See also
NimSessionNewList

nimSessionGetNewCon

Signature

```
NIMSESS      *nimSessionGetNewCon(NIMSESS *nims);
```

Description
Internal only. Get a new client connection (session) from the server session.

Parameters

NIMSESS *	*nims*	NimBUS server session.

Returns
Pointer to a new session or NULL if there is no new client session.

See also
nimSession

nimSessionGetMsg

Signature

```
NIMMSG       *nimSessionGetMsg(NIMSESS *nims);
```

Description
Internal. Get a new message from a session. You must free the message after use.

Parameters

NIMSESS *	*nims*	Pointer to a session

Returns
Pointer to a new message or NULL if none is available.

See also
nimMsgFree

nimSessionSetData

Signature

```
int   nimSessionSetData(NIMSESS *nims, void *data);
```

Description
Used by server applications (probes) to attach session specific data to a client session. The server must free the data before the session is freed.

Parameters

NIMSESS *	nims	Pointer to a session
void *	data	Pointer to session data

Returns
NIME_OK

See also
nimSessionGetData

nimSessionGetData

Signature

```
void* nimSessionGetData(NIMSESS *nims);
```

Description
Return pointer to the session data. Used in session callbacks to access the session data.

Parameters

NIMSESS *	*nims*	Pointer to a session

Returns
Pointer to the session data or NULL if not available.

See also
nimSessionSetData

nimSessionGetLastError

Signature

```
int   nimSessionGetLastError(NIMSESS *nims);
```

Description
Returns the last error code on the session.

Parameters

NIMSESS *	*nims*	Pointer to a session

Returns
Last error code on the session.

See also
nimError2Txt

nimSessionConnect

Signature

```
int   nimSessionConnect(NIMSESS *nims, char *szHost, int iPort, int secWait);
```

Description
Connect to a server (probe) with a given host name or IP address and on the given port number.

Parameters

NIMSESS *	nims	Pointer to a session
char *	szHost	Host to connect to
int	iPort	Port to connect to
int	secWait	Timeout

Returns
NIME_OK or an error code.

See also
nimSessionNew

nimSessionServer

Signature

```
int   nimSessionServer(NIMSESS *nims, int iPort);
```

Description
Create a server session on an existing session.

Parameters

NIMSESS *	nims	Pointer to a session
int	iPort	Port to listen on

Returns
NIME_OK or an error code

See also
nimSessionNew

nimSessionSetDatagram

Signature

```
int   nimSessionSetDatagram(NIMSESS *nims, int defaultLocalPort);
```

Description
This function will set this session type to datagram. (UDP) for both sending and receiving.
Example of broadcasting:

```
nims = nimSessionNew(); nimSessionSetDatagram(nims,8000,0);
nimSessionConnect(nims,"193.71.55.255",8000) nimSessionSend(nims,"gethub",pds);
```

Parameters

NIMSESS *	nims	Pointer to a session
Int	defaultLocalPort	Port to listen on

Returns
None

See also
nimSessionNew

nimSessionGetOption

Signature

```
int nimSessionGetOption (NIMSESS *nims)
```

Description
Returns options for a session.

Parameters

NIMSESS *	*nims*	Session to look up options from.

nimSessionSetOption

Signature

```
void nimSessionSetOption (NIMSESS *nims, int iOption)
```

Description
Allows you to replace the options for a given session.

Parameters

NIMSESS *	nims	Session to change
int	iOption	Options that are valid for this session after the function call

nimSessionIsConnected

Signature

```
int   nimSessionIsConnected(NIMSESS *nims);
```

Description
Check if a session is connected.

Parameters

NIMSESS *	*nims*	Pointer to a session

Returns
NIME_OK or error code.

See also
nimSessionConnect

nimSessionSpooler

Signature

```
NIMSESS     *nimSessionSpooler();
```

Description
Create a session connected to the default spooler. This convenience function will create a session to the default spooler. Use this function when you expect to load many messages onto the spooler.

Parameters
N/A

Returns
NIMSESS * on success or a NULL if an error occurred.

See also
nimSessionNew, nimSessionAlarm, nimSessionPostMessage.

nimSessionController

Signature

```
NIMSESS      *nimSessionController();
```

Description
Create a session connected to the default controller. This convenience function will create a session to the default controller.

Parameters
N/A

Returns
NIMSESS * on success or NULL if an error occurred.

See also
nimSessionNew.

nimSessionHub

Signature

```
NIMSESS      *nimSessionHub(char *hubaddress);
```

Description
Create a session connected to the hub. This convenience function will create a session to the named hub. If 'hubaddress' is NULL, then the NIMV_HUBADDR is used.

Parameters

char *	*hubaddress*	The hub address, to which the session is to be

Returns
NIMSESS * on success or NULL if an error occurred.

See also
nimSessionNew.

nimSessionAlarm

Signature

```
int   nimSessionAlarm(NIMSESS *nims,char *message, int pri, char *suppression, char
*subsys,char *source, char **nimid);
```

Description
Send a NimBUS Alarm message on the session. This function behaves exactly like the nimAlarm function, except that a session is required prior to sending the alarm.

Parameters

NIMSESS *	nims	A session connected to a spooler
char *	message	The alarm message to be sent on the session.
int	pri	Severity
char *	suppression	Suppression key
char *	subsys	Subsystem (e.g. 1.1.3)
char *	source	Alarm source, NULL = hostname
char **	nimid	Identifier of the new alarm

Returns
NIME_OK or an error code.

See also
nimSessionNew, nimSessionSpooler, nimAlarm.

nimSessionDispatch

Signature

```
int   nimSessionDispatch (NIMSESSLIST *list,int millisec, int breakOnTimeout);
```

Description
The session dispatcher is the engine of the probe. When you have prepared the session by setting up command callbacks (nimSessionAddCallback) and the standard callbacks (nimSessionAddStdCallback) most of the probe logic will be handled by the dispatcher with the exception of what is done on timeouts.

Parameters

NIMSESSLIST *	list	Pointer to a session list
int	millisec	Timeout value
int	breakOnTimeout	Return from function on timeout?

Returns
NIMSW_TIMEOUT is returned on timeout if 'breakOnTimeout' is set. All other exit codes are error codes like: NIMSW_EXIT, NIMSW_RESTART and NIMSW_ERROR.

See also
nimSessionNewList, nimSesionAddCallback, NimSessionAddStdCallback

nimSessionAddStdCallback

Signature

```
int   nimSessionAddStdCallback(NIMSESS *nims,NIMINFO *info,void *cbdata);
```

Description
Add session standard callback functions. This function will add the standard callback functions for status, list, stop and restart. The NIMINFO struct also contains name, version and company information that are returned by the *_status* command.

Parameters

NIMSESS *	nims	Session to attach the standard callback on
NIMINFO *	info	Session info struct to add.
void *	cbdata	0

Returns
NIME_OK on success or an error code on failure.

See also
nimSessionAddStdCallback.

nimSessionAddCallback

Signature

```
Int nimSessionAddCallback (NIMSESS *nims,char *cmd, int (*func)(NIMCB *cb), char
*fmt, int sec_rights);
```

Description
Add command callback function on session. This function will register a user-defined callback to the session (or session list) for the command 'cmd' provided by the caller. The callback function prototype is determined by the format 'fmt' parameter. The format string is defined by pdsScanf(3), and enables the programmer to specify the parameter order and type to the callback function. A user specific 'cbdata' assocated with the callback may be added by the nimSessionAddCbData.

Parameters

NIMSESS *	nims	Pointer to a session.
char *	cmd	Command name.
int *	func(NIMCB * cb)	Callback function
char *	fmt	Callback parameter format string
int	sec_rights	Required permission

Returns
NIME_OK on success or an error code on failure.

See also
nimSessionNew, nimSessionServer.

nimSessionCallbackData

Signature

```
int   nimSessionCallbackData (NIMSESS *nims,char *cmd, void *cbdata);
```

Description
Add callbackdata to 'cmd' callback. This function will attach the data to the callback structure associated with 'cmd'.

Parameters

NIMSESS *	nims	Pointer to a session
char *	cmd	Command to add the data to
void *	cbdata	Pointer to callback data

Returns
NIME_OK on success. NIME_ERROR, NIME_INVAL if errors occur.

See also
nimSessionAddCallback.

nimSessionPostMessage

Signature

```
int nimSessionPostMessage(NIMSESS *nims, char *subject,int pri, char *suppression,
PDS *data, char **nimid);
```

Description
Does the same as nimPostMessage but requires a session connected to a spooler. Use nimSessionSpooler or nimSession to obtain the session.

Parameters

NIMSESS *	nims	Pointer to a spooler session
char *	subject	Message subject
int	pri	Use 0
char *	suppression	Suppression key, normally NULL
PDS *	data	Data to post
char **	nimid	NimBUS identifier for the posted message

Returns
NIME_OK or an error code.

See also
nimSessionSpooler, nimSession

nimSessionAddSessionCallback

Signature

```
int   nimSessionAddSessionCallback (NIMSESS *nims, int (*func)());
```

Description
Add reply callback function on session. This function will register a user-defined callback to the session for the any reply provided by the caller.

The callback function prototype is

```
int func(NIMSESS *nims, void *data, int what, NIMMSG *nmsg)
```

Values for what = NIMSW_REPLY - reply is in nmsg
= NIMSW_SCONNECT - nims is connected
= NIMSW_STIMEOUT - request/connection timedout
= NIMSW_ERROR - error/disconnect

Valid return values are Zero

Parameters

NIMSESS *	nims	Pointer to a session
int *	(*func)()	Pointer to the dallback function

Returns
NIME_OK on success. NIME_ERROR if an error occurred.

See also
nimSessionNew, nimSessionServer.

nimSessionClearCallback

Signature

```
int   nimSessionClearCallback(NIMSESS *nims);
```

Description
This function will disconnect any callback information from this session.

Parameters

NIMSESS *	*nims*	Pointer to a session.

Returns
NIME_OK on success. NIME_ERROR if an error occurs.

See also
nimSessionAddCallback

nimUnRegisterProbe

Signature

```
int   nimUnRegisterProbe(char *probename);
```

Description
This function will unregister probename at the controller.

Parameters

char *	*probename*	Probe name to unregister

Returns
NIME_OK or an error code.

See also
nimRegisterProbe.

nimSessTraceLevel

Signature

```
int   nimSessTraceLevel(NIMSESS *nims,int level);
```

Description
Sets the debug level for low level logging of the session.

Parameters

NIMSESS *	nims	Session for which to set trace level
int	level	Trace level

See Also
N/A

nimSessErrorLevel

Signature

```
Int nimSessErrorLevel (NIMSESS *nims, int level)
```

Description
No longer used, but available for backwards compatibility.

nimSessResponseTimeout

Signature

```
Int nimSessResponseTimeout (NIMSESS *nims, int level)
```

Description
No longer used, but available for backwards compatibility.

Header Functions

The following functions are defined in the header file, but are not for use outside the library:

- `Int nimSessionWait(NIMSESS *nims, int millWait, NIMSESS **psnims);`
- `Int nimSessionWaitReply(NIMSESS *nims, int millWait, int *status, PDS **dpds);`
- `Int nimSessionWaitMsg(NIMSESS *nims, int secWait, NIMMSG **pnmsg);`
- `NIMSESS*  nimSessionGetNewCon(NIMSESS *nims);`
- `int nimSessionAConnect(NIMSESS *nims, char *szHost, int iPort, int secWait);`
- `int nimSessionARequest(NIMSESS *nims, char *cmd, PDS *data, int secWait);`
- `int nimSessionSend(NIMSESS *nims, char *cmd, PDS *data);`
- `int nimSessionSendRaw(NIMSESS *nims, char *buf, size_t nc);`
- `int  nimMsgGetRaw(NIMMSG *nmsg, void *buf, size_t *size);`
- `void nimSetLocalIp (char *pchIp);`

Miscellaneous Methods

nimFindAsTable

Signature

```
int nimFindAsTable(PDS *pdsFilter, int iType, char ***pppchOut);
```

Description
This is a wrapper around nimFindAsFunc(). cslLineInsert() is used to fill pppchOut.

Parameters

PDS *	pdsFilter	Contains the "filter" used to specify the criterias. Available fields in the "filter" PDS are: domain, hubip, hubname, hubversion, robotip, robotname, robotversion, osmajor, osminor, osuser1, osuser2, probename, pkg_name, pkg_version, active. In addition the int "timeout" can be set for nimRequest (default=10).
int	iType	Available types: NIMF_HUB, NIMF_ROBOT, NIMF_PROBE.
char ***	pppchOut	Gets filled with a NULL terminated string table containing the NimBUS addresses requested. The string should be de-allocated using cslTblFree().

Returns
NIME_OK, NIME_ERROR or NIME_NOENT (if nothing matches criteria).

See also
nimFindAsPds, nimFindCreateFilter

nimFindAsPds

Signature

```
int nimFindAsPds (PDS *pdsFilter, int iType, PDS **pdsOut);
```

Description
This is a wrapper around nimFindAsFunc(). pdsPutTable() is used to fill pdsOut.

Parameters

PDS *	pdsFilter	Contains the "filter" used to specify the criterias. Available fields in the "filter" PDS are: domain, hubip, hubname, hubversion, robotip, robotname, robotversion, osmajor, osminor, osuser1, osuser2, probename, pkg_name, pkg_version, active. In addition the int "timeout" can be set for nimRequest (default=10).
int	iType	Available types: NIMF_HUB, NIMF_ROBOT, NIMF_PROBE.
PDS**	pdsOut	Gets filled with a table of NimBUS addresses use pdsTableGet(*pdsOut,"addr",&myaddr) to read into (char *)myaddr. The PDS must be removed by the calling function.

Returns
RETURNS: NIME_OK, NIME_ERROR or NIME_NOENT (if nothing matches criteria).

See also
nimFindAsFunc(), nimFindCreateFilter() for list of available filters.

nimFindAsFunc

Signature

```
int nimFindAsFunc(PDS *pdsFilter, int iType, void *vpOut, int (*pFunc)(char *, void
*));
```

Description
Run a find on all NimBUS probes that returns all addresses matching the find filter. When an address matches the function will be called with the address as parameter one.

Parameters

PDS *	pdsFilter	Contains the "filter" used to specify the criterias.
int	iType	Available types: NIMF_HUB, NIMF_ROBOT, NIMF_PROBE.
void *	vpOut	Used as second argument to *pFunc* if your function needs to fill in a block of data (use NULL if it does not).
int *	PFunc(char *, void *)	Function pointer. Allows you to specify your own function to be run upon every matched address.

Returns
NIME_OK, NIME_ERROR or NIME_NOENT (if nothing matches criteria).

See also
nimFindAsTablefor list of available filters.

nimFindCreateFilter

Signature

```
PDS *nimFindCreateFilter(char *domain, char *hubip, char *hubname, char *hubversion,
char *robotip, char *robotname, char *robotversion, char *osmajor, char *osminor,
char *osuser1, char *osuser2, char *probename, char *pkg_name, char *pkg_version,
char *group, char *active);
```

Description
Create a filter that is used in all the find functions to match the find criteria. Use NULL to if there is no match criteria for the parameter.

Parameters

char *	domain	Corresponds to a filter possibility in the *nimFind* family of functions.
char *	hupip	Corresponds to a filter possibility in the *nimFind* family of functions.
char *	hubname	Corresponds to a filter possibility in the *nimFind* family of functions.
char *	hubversion	Corresponds to a filter possibility in the *nimFind* family of functions.
char *	robotip	Corresponds to a filter possibility in the *nimFind* family of functions.
char *	robotname	Corresponds to a filter possibility in the *nimFind* family of functions.
char *	robotversion	Corresponds to a filter possibility in the *nimFind* family of functions.
char *	osmajor	Corresponds to a filter possibility in the *nimFind* family of functions.
char *	osminor	Corresponds to a filter possibility in the *nimFind* family of functions.
char *	osuser1	Corresponds to a filter possibility in the *nimFind* family of functions.
char *	osuser2	Corresponds to a filter possibility in the *nimFind* family of functions.

char *	pkg_name	Corresponds to a filter possibility in the *nimFind* family of functions.
char *	pkg_version	Corresponds to a filter possibility in the *nimFind* family of functions.
char *	group	Corresponds to a filter possibility in the *nimFind* family of functions.
char *	active	Corresponds to a filter possibility in the *nimFind* family of functions.

Returns
Pointer to a PDS containing the filter values.

See also
nimFindAsFunc(), nimFindAsPds(), nimFindAsTable(), nimFindSetTimeOut()

nimFindSetTimeOut

Signature

```
int nimFindSetTimeOut(PDS **pdsFilter, int timeout);
```

Description
Set timeout value for the find in the filter PDS.

Parameters

PDS *	pdsFilter	Address of a PDS containing the filter for the *nimFind* functions.
int	timeout	Timeout used for nimRequest calls.

Returns
NIME_OK, NIME_ERROR

See also
nimFindAsFunc(), nimFindAsPds(), nimFindAsTable(), nimFindCreateFilter().

nimFree

Signature

```
Void nimFree(void *f)
```

Description
Wrapper over the C function free() with a check to avoid NULL pointers.

Parameters

void	f	Pointer to memory area to be freed

Returns
N/A

nimGetVarStr

Signature

```
int   nimGetVarStr(int what, char *data, int dlen);
```

Description
Get the requested API value. Typically used for fetching the current hub, spooler or controller IP.

Parameters

int	what	API value to get
char *	data	Buffer to return value in
int	dlen	Buffer size

Returns
NIME_OK or error code.

See also
nimSetVarStr, nimGetVarInt

nimGetNameToIp

Signature

```
int   nimGetNameToIp(char *name, char *ip, int iplen, int *port);
```

Description
Look up a NimBUS address to find the IP address and port number to connect to, used together with nimSessionConnect or nimSession. Use nimNamedSession instead.

Parameters

char *	name	NimBUS address
char *	ip	IP address found for name
int	iplen	Size of IP buffer
int *	port	Port found for name

Returns
NIME_OK or an error code.

See also
nimNamedSession, nimSessionConnect, nimSession

nimGetVarInt

Signature

```
int nimGetVarInt(int what, int *data);
```

Description
Return the requested API value. E.g. what = NIMV_SPOOLPORT returns the local spooler port.

Parameters

int	what	API value to get
int *	data	The value

Returns
NIME_OK or an error code.

See also
nimGetVarInt, Nimfree, NimGetVarStr

nimPDS2section

Signature

```
int   nimPDS2section(char *file, PDS *in);
```

Description
Write the content of a PDS with configuration data to a file. Use the file name from the PDS if 'file' is NULL.
Parameters

char *	file	Output file name
PDS *	in	PDS containing config data.

Returns
NIME_OK or an error code.

See also
nimSection2PDS

nimSection2PDS

Signature

```
PDS   *nimSection2PDS(char *file, char *section, PDS *inout);
```

Description
Converts the content of a NimBUS formatted configuration file to PDS. If the input PDS is NULL a new PDS is created, if it is not NULL data is appended to the existing PDS. This function is often used together with nimPDS2Section.

Parameters

char *	file	Input file name
char *	section	Section in configuration file (.cfg)
PDS *	inout	Result PDS

Returns
The result PDS or NULL on failure.

See also
nimPDS2section

nimSetVarInt

Signature

```
int   nimSetVarInt(int what, int data);
```

Description
Set the named API variable. Typically used to change the default spooler port.

Parameters

int	what	API value to set
int	data	New value

Returns
NIME_OK or an error code.

See also
N/A

nimSetTrapHandler

Signature

```
void  nimSetTrapHandler(int (*func)());
```

Description
Sets the terminate trap handler. The function will try to trap terminate interrupts and call func, so the application can clean up and exit. Only one function can be defined per process.

Parameters

int *	*(func)()*	Callback function when a signal is trapped

Returns
None

See also
N/A

nimSetVarStr

Signature

```
int nimSetVarStr(int what, char *data);
```

Description
Set a new API value. Typically used to change default spooler IP.

Parameters

int	what	API value to set
char *	data	New value

Returns
NIME_OK or error code.

See also
Nimfree, nimvarstr.

nimSuppToStr

Signature

```
char *nimSuppToStr(char *szSupp, int nBytes, int bHold, int iNumber, int iSeconds,
char *szSuppressionKey);
```

Description
Create a suppression string for nimAlarm. The suppression buffer ('szSupp') must be at least 12 bytes ('nBytes'). The 'bHold' flags decides if the first 'iNumber' alarms should be suppressed before sending an alarm or if the first alarm should be sent before suppressing. 'iSeconds' is the time a suppression is valid. If the time is out, the next alarm will be sent and the suppression counter reset for that alarm.

Parameters

char *	szSupp	Suppression string.
int	nBytes	Size of szSupp.
int	bHold	Send and then suppress.
int	iNumber	Suppress n messages.
int	iSeconds	Within n seconds.
char *	szSuppressionKey	Suppression key.

Returns
Pointer to szSupp on success. Pointer to NULL if an error occures.

See also
nimAlarm.

nimNextTime

Signature

```
Time_t nimNextTime(const char *pchTimeSpecification)
```

Description
The function returns the time_t of the next matching time specification.

Parameters

const char *	pchTimeSpecifi	Time specification in the form of hh:mm:ss

See also
N/A

nimTimerStart

Signature

```
Void nimTimerStart (NIMTIMER *t);
```

Description

Sets the start time in the NIMTIMER object.

Parameters

NIMTIMER	*t	Pointer to NIMTIMER object

See also
N/A

nimTimerStop

Signature

```
Void nimTimerStop(NIMTIMER *t);
```

Description
Sets the stop time in the NIMTIMER object.

Parameters

NIMTIMER	*t	Pointer to NIMTIMER object

See also
N/A

nimTimerDiff

Signature

```
Long nimTimerDiff(NIMTIMER *t);
```

Description
Creates a lap-time (temporarily) if not stopped with nimTimerStop already, and computes the time difference in milliseconds between the start and stop (or current) time.

Parameters

NIMTIMER	*t	Pointer to NIMTIMER object

See also
N/A

nimTimerDiffSec

Signature

```
Double nimTimerDiffSec(NIMTIMER *t)
```

Description
Creates a lap-time (temporarily) if not stopped with nimTimerStop already, and computes the time difference in seconds between the start and stop (or current) time.

Parameters

NIMTIMER	*t	Pointer to NIMTIMER object

Crypto methods

nimCheckPassword

Signature

```
int   nimCheckPassword(const char *md5, const char *password);
```

Description
Check a password against the one-way encoded MD5 string.

Parameters

const char *	md5	MD5 one-way string
const char *	password	The password

Returns
NIME_OK on success or an error code on failure.

See also
nimNewPassword

nimDecryptString

Signature

```
char  *nimDecryptString(const unsigned char *key,const int keylen, const char *data);
```

Description
The input string must be encrypted by nimEncryptString and the key must be the same. The result must be freed after use.

Parameters

Const unsigned char*	*key*	Decryption key
const int	keylen	Key length
const char*	*data*	String to decrypt

Returns
The decrypted data or NULL on failure.

See also
nimEncryptString

nimEncryptString

Signature

```
char  *nimEncryptString(const unsigned char *key,const int keylen, const char
*string);
```

Description
Will encrypt the input string using the Twofish algorithm and the encryption key. The function returns a base64 encoded and encrypted string. The result must be freed after use.

Parameters

const unsigned char *	key	Encryption key
const int	keylen	Encryption key length
const char*	string	String to encrypt

Returns
A string or NULL on failure.

See also
nimDecryptString

nimMD5Sum

Signature

```
int   nimMD5Sum(const char *file, unsigned char *digest);
```

Description
Calculate the MD5 checksum for a given file. The digest must be of size MD5_DIGEST (16 bytes).

Parameters

const char *	file	Input file
char *	digest	MD5 result

Returns
File size or 0 on error.

See also
N/A

nimNewPassword

Signature

```
char  *nimNewPassword(const char *password);
```

Description
Internal. Create a MD5 based one-way password entry that can be used for checking a password later.

Parameters

const char *	*password*	A password

Returns
The MD5

See also
nimCheckPassword

QoS methods

nimQoSCreate

Signature

```
NIMQOS      *nimQoSCreate(char *pchQoS, char *pchSource, long lSampleRate, long
lSampleMaxValue);
```

Description
Create a QoS handle for sending data sampled on regular interval. Used together with the nimQoSSend methods. You must call nimQoSFree to release all resources allocated in the handle when you are done.

Parameters

char *	pchQoS	QoS name
char *	pchSource	QoS source (NULL = hostname)
long	lSampleRate	Sample rate.
long	lSampleMaxValue	Max value if any (-1 = no max value).

Returns
NIMQOS handle on success or NULL on failure.

See also
NimQoSFree, nimQosSendValue

nimQoSCreateAsynch

Signature

```
NIMQOS *nimQoSCreateAsynch(char *pchQoS, char *pchSource, double dSampleMaxValue);
```

Description
Creates an asynchronous QoS object, which is one that is not collected on an interval like regular QoS objects.

Parameters

char	*pchQoS	QoS name
char	*pchSource	QoS source
double	dSampleMaxValue	Maximum value that is valid for this QoS

See also
nimQoSCreate, nimQoSFree

nimQoSDefinition

Signature

```
int   nimQoSDefinition(char *pchName, char *pchGroup, char *pchDescription, char
*pchUnit, char *pchUnitShort, int bMax, int bBool);
```

Description
Send a new QoS definition that describes a new QoS type. Typically sent once when the probe starts.

Parameters

char *	pchName	QoS name on the form QOS_<SOMETHING>
char *	pchGroup	QoS group on the form QOS_<SOMETHING>
char *	pchDescription	QoS description
char *	pchUnit	QoS unit type. E.g. Milliseconds, Megabytes
char *	pchUnitShort	QoS unit short form. E.g. ms, MB
int	bMax	Has max value.
int	bBool	Is Boolean (reports 0 or 1 only)

Returns
NIME_OK on success or an error code on failure.

See also
nimQoSMessage

nimQoSSEndDefinition

Signature

```
int nimQoSSEndDefinition(char *pchName, char *pchGroup, char *pchDescription, char
*pchUnit, char *pchUnitShort, int iFlags);
```

Description
This function replaces the old nimQoSDefinition function, and sends a QOS_DEFINITION message to the NimBUS. The flags can be one of NIMQOS_DEF_NONE, NIMQOS_DEF_BOOLEAN, NIMQOS_DEF_HASMAX, NIMQOS_DEF_ASYNCH, NIMQOS_DEF_REDEFINE.

Parameters

Char	*pchName	Tthe name of this QoS
Char	*pchGroup	The group this QoS belongs to
Char	*pchDescription	The description of this QoS
Char	*pchUnit	The unit of measurement this QoS value has (e.g. Megabytes)
Char	*pchUnitShort	The short version of pchUnit above, used in graphs (e.g. MB)
int	iFlags	flags for different types of values.

Note
nimQoSDefinition should be modified to point to the new function and listed as deprecated!

nimQoSFree

Signature

```
void  nimQoSFree(NIMQOS *Handle);
```

Description
Release the resources allocated in the handle.

Parameters

NIMQOS *	*Handle*	QoS handle

Returns
N/A

See also
nimQoSCreate

nimQoSGetSource

Signature

```
Char *nimQoSGetSource(NIMQOS *Handle);
```

Description
Returns the source field from the Handle.

Parameters

NIMQOS *	*Handle*	QoS handle

nimQoSGetTimer

Signature

```
unsigned long nimQoSGetTimer(NIMQOS *Handle);
```

Description
Get the number of milliseconds since nimQoSStart was called. Will return the number of milliseconds up to now even if nimQoSStop was not called.

Parameters

NIMQOS *	*Handle*	QoS handle

Returns
Time in milliseconds.

See also
nimQoSSetSampletime, nimQoSStart, nimQoSStop

nimQoSMessage

Signature

```
int   nimQoSMessage(char *pchQoS,char *pchSource, char *pchTarget, long lSampleTime,
double dSampleValue,double dSamlpeStdev, long lSampleRate, long lSampleMax);
```

Description
Send a raw QoS message. Use nimQoSCreate, nimQoSSendxxx and nimQoSFree instead.

Parameters

char *	pchQoS	QoS name
char *	pchSource	QoS source
char *	pchTarget	QoS target
Long	lSampleTime	Sample time (EPOC)
double	dSampleValue	Sample value
double	dSampleStdev	Sample standard deviation
long	lSampleRate	Sample rate in seconds
long	lSampleMax	Sample max value or –1 if not applicable

Returns
NIME_OK on success or an error code on failure.

See also
nimQosDefinition

nimQoSPostMessage

Signature

```
Int nimQoSPostMessage (NIMQOS *Handle, char *pchTarget, double dSampleValue, double
dSampleStdev);
```

Description
The nimQoSSend functions are wrappers to this function which does the sending of QoS messages.

Parameters

NIMQOS *	Handle	QoS handle
char	*pchTarget	QoS target
double	dSampleValue	value of this QoS measurement
double	dSampleStdev	this QoS measurement's standard deviation

See also
nimQoSSendValue, nimQoSSendNull, nimQoSSendTimer, nimQoSSendValueStdev

nimQoSSendTimer

Signature

```
int   nimQoSSendTimer(NIMQOS *Handle, char *pchTarget);
```

Description
Send a QoS message with the time in milliseconds since nimQoSStart was called.

Parameters

NIMQOS *	Handle	QoS handle.
char *	pchTarget	QoS target.

Returns
NIME_OK on success or an error code on failure.

See also
nimQoSCreate, nimQoSCreate, nimQoSStop

nimQoSSendValue

Signature

```
int   nimQoSSendValue(NIMQOS *Handle, char *pchTarget, long lValue);
```

Description
Send a QoS message with a given value.

Parameters

NIMQOS *	Handle	QoS handle
char *	pchTarget	QoS target
long	lValue	Sample value

Returns
NIME_OK on success or an error code on failure.

See also
nimQoSSendValueStdev

nimQoSSendValueStdev

Signature

```
int   nimQoSSendValueStdev(NIMQOS *Handle, char *pchTarget, double dValue, double
dStdev);
```

Description
Send a QoS message with a given double value and the sample standard deviation if applicable.

Parameters

NIMQOS *	Handle	QoS handle
char *	pchTarget	QoS target
double	dValue	Sample value
double	dStdev	Standard deviation. Typical 0

Returns
NIME_OK on success or an error code on failure.

See also
nimQoSSendValue

nimQoSSendNull

Signature

```
int   nimQoSSendNull(NIMQOS *Handle, char *pchTarget);
```

Description
Send a QoS message with a NULL sample. Used to indicate that the target was unavailable for monitoring.

Parameters

NIMQOS *	Handle	QoS handle
char *	pchTarget	QoS target

Returns
NIME_OK on success or an error code on failure.

See also
nimQoSCreate

nimQoSSetSampletime

Signature

```
Void nimQoSSetSampletime(NIMQOS *Handle, time_t tSampleTime);
```

Description

Overrides the sampletime in the Handle object. The sampletime is normally set to when the handle object is created, however you may wish to override this to get more exact timing.

Parameters

NIMQOS	*Handle	handle to QoS object
time_t	t	SampleTime sampletime value set in Handle

nimQoSSourceIsSet

Signature

```
Int nimQoSSourceIsSet(viod);
```

Description
(Internal function).

nimQoSHostname

Signature

```
Char *nimQoSHostname(char *pch, size_t len);
```

Description
(Internal function) Returns hostname used as soure in QoS messages.

nimQoSStart

Signature

```
void  nimQoSStart(NIMQOS *Handle);
```

Description
Start the internal timer.

Parameters

NIMQOS *	*Handle*	QoS handle

Returns
N/A

See also
nimQoSStop, nimQoSSendTimer

nimQoSStop

Signature

```
void  nimQoSStop(NIMQOS *Handle);
```

Description
Stop the internal timer.

Parameters

NIMQOS *	*Handle*	QoS handle

Returns
N/A

See also
nimQosSetSampletime, nimQosSourceIsNet,

Chapter 2: PDS (Portable Data Stream) library

The Portable Data Stream (PDS) library contains a set of functions meant to manipulate machine independent data represented by the PDS object. This object may in turn be used, for instance, to send network independent information from client to server. The data stream built by PDS is an ASCII stream containing elements built using the following format:

KEY_NAME	PDS_TYPE	DATA_SIZE	DATA

pdsCopy

Signature

```
PDS *pdsCopy(PDS *source, PDS **destination);
```

Description
Creates a copy of the *Source* PDS previously created with pdsCreate() by allocating new memory and copy the PDS contents.

Parameters

PDS *	source	A pointer to the existing PDS you want to copy.
PDS **	destination	The address of a PDS pointer, which will be set to point to the copy of the source PDS. This parameter is optional.

Returns
A pointer to the copy of the source PDS.
On allocation error, NULL is returned and pdsError(Source) will return PDS_ERR_MALLOC.

See also
pdsCreate(), pdsPut(), pdsGet() and pdsDelete().

pdsCount

Signature

```
int   pdsCount(PDS *pds);
```

Description
This convenience function will return the number of PDS elements in the specified PDS object.

Parameters

PDS *	*pds*	Pointer to an existing PDS.

Returns
The number of elements found in *pds*.

See also
pdsPut(), pdsGet() and pdsCreate().

pdsCreate

Signature

```
PDS *pdsCreate();
```

Description
This function is a convenience function on top of pdsCreateSize(), and will create a PDS with data size 1024 bytes.

Parameters
None

Returns
A pointer to the new PDS, or NULL.

See also
pdsPut(), pdsGet(), pdsDelete() and pdsCreate().

pdsCreateSize

Signature

```
PDS *pdsCreateSize(intsize);
```

Description
This function creates a PDS object and allocates size space for data.

Parameters

PDS *	source	A pointer to the existing PDS you want to copy.
PDS **	destination	The address of a PDS pointer, which will be set to point to the copy of the source PDS. This parameter is optional.

Returns
A pointer to the new PDS, or NULL.

See also
pdsPut(),pdsGet(),pdsDelete() and pdsCreate().

pdsDelete

Signature

```
int pdsDelete(PDS *pds);
```

Description
This function will free the data related to the provided PDS handle previously created by pdsCreate().

Parameters

PDS *	*pds*	A pointer to an existing PDS you want to delete.

Returns
PDS_ERR_NONE on success and PDS_ERR if the *pds* pointer does not point to an existing PDS.

See also
pdsCreate()

pdsDump

Signature

```
int pdsDump(PDS *pds);
```

Description
This function will run through the PDS, element for element and print the name, type, size and some of the data of the current PDS.

Parameters

PDS *	*pds*	A pointer to an existing PDS.

Returns
PDS_ERR_NONE on success and PDS_ERR on error.

See also
pdsCreate(),pdsPut() and pdsPrintf().

pdsExpand

Signature

```
char *pdsExpand(PDS *pds, char *format, ...);
      /* optional paramater: char *missing_value */
```

Description
This function will expand the variables prefixed by '$' in the PDS stream.

Parameters

PDS *	pds	A pointer to an existing PDS.
char *	format	String containing variable names prefixed with '$'.
char *	missing_value	Optional parameter.

Returns
On success, a pointer to an allocated buffer, and on error NULL.

pdsFreeTable

Signature

```
void pdsFreeTable(PDStype t, void **table);
```

Description
This function will free data allocated by the PDS library. Supported types: PDS_PPCH, PDS_PI, PDS_PL and PDS_PPDS.

Parameters

PDStype	t	The type of data to be freed.
void **	table	The actual data to be freed.

pdsGet

Signature

```
int pdsGet(PDS *pds, PDStype t, char *key, void *d);
```

Description
This function will get the next element pointed to by the 'get' pointer within the PDS object.

Parameters

PDS *	pds	A pointer to an existing PDS.
PDStype	t	The type of element to fetch.
char *	key	The name, which refers to the element being fetched.
void *	d	Address to a pointer, which will be set to the element.

Returns
The size will be returned for PDS_VOID and PDS_PDS, otherwise 0 on success and 1 on error. Use pdsError to get the actual error code.
The error code returned can be:
PDS_ERR_ILLTYPE - Protocol error (type error). PDS_ERR_ILLARG - Illegal input value.
PDS_ERR_MALLOC - Unable to allocate memory. PDS_ERR_BOUNDS - Data transmission error.
PDS_ERR_NOMATCH - No matching key in PDS.

Example:

```
PDS *pds;
int i;
// ******************************************
// GET INTEGER FROM STREAM, NAMED/TAGGED VALUE. pdsGet(pds, PDS_INT, "value", i, 0);
// ************************************************
// GET CURRENT INTEGER FROM STREAM, IGNORING THE NAME.
pdsGet(pds, PDS_INT, NULL, &i);
// Get next data in stream
```

All tables PI, PPI, PCH, PPCH, PF, PPF are Null terminated.

See also
pdsCount(), pdsCreate() and pdsCount().

pdsGetData

Signature

```
void *pdsGetData(PDS *pds, char *key, PDStype t);
```

Description
This function will extract the data acc. to key (or NULL if next element in PDS), and convert it according to the type parameter. The data returned must be freed by the caller. The following type specifiers will return: PDS_PCH - Pointer to null-terminated string. PDS_INT - Pointer to number (int) PDS_FLOAT - Pointer to double (double)

Parameters

PDS *	pds	A pointer to an existing PDS.
char *	key	The name of the element to fetch.
PDStype	t	The type of the element to fetch.

Returns
The element fetched or NULL. On NULL return, use pdsError () to get the error code.

See also
pdsPut() and pdsGet().

pdsGetNext

Signature

```
int pdsGetNext(PDS *pds, char **key, PDStype *type, long *size, void **data);
```

Description
This function will use the current 'get' position as the starting point for element extraction. Please use the pdsRewind() function prior to the first pdsGetNext() call. Note that if no pdsGet() or any of the pdsGet convenience functions (e.g. pdsGet_INT) have been used, then the get pointer is initiated to the start of the PDS buffer.

Parameters

PDS *	pds	A pointer to an existing PDS.
char **	key	Pointer to the address to get the element key.
PDStype *	type	Pointer to a variable, which gets the element type.
long *	size	Pointer to a variable, which gets the element size
void **	data	Pointer to an address, which gets the actual element.

Returns
PDS_ERR_NONE on success or PDS_ERR on error. Use pdsError() to get the specific error code.

See also
pdsGet(), pdsScanf(), pdsCreate() and pdsRewind()

pdsCfgRead

Signature

```
int   pdsCfgRead(char *filename, PDS **pds);
```

Description
This function will parse a given .cfg file into a PDS structure where each section becomes a PDS with the full section path as its name. *pds* will be created, even if the configuration file is empty.

Parameters

char *	filename	The name of the configuration file to be read
PDS **	pds	The address of a PDS pointer, which will be set to the allocated PDS.

Returns
On success PDS_ERR_NONE, on error PDS_ERR_ILLARG – illegal arguments or PDS_ERR_NOMATCH – no such file.

See also
pdsCreate()

pdsGet_PPDS

Signature

```
int pdsGet_PPDS(PDS *pds, char *key, PDS ***pppPds);
```

Description
This convenience function will extract the PDS table named key (or the next element if key is NULL), and return a pointer to an allocated table of PDS structures provided by this API.

Parameters

PDS *	pds	A pointer to an existing PDS.
char *	key	The key to the element to be returned.
PDS ***	pppPds	A pointer to a PDS table pointer.

Returns
On success PDS_ERR_NONE, on error PDS_ERR. The specific error code is returned with pdsError().

See also
pdsGet(), pdsScanf() and pdsCreate()

pdsGet_PDS

Signature

```
int pdsGet_PDS(PDS *pds, char *key, PDS **ppds);
```

Description
This convenience function will extract the PDS element named key (or the next element if key is NULL), and return a pointer to an allocated PDS structure provided by this API.

Parameters

PDS *	pds	A pointer to an existing PDS.
char *	key	The key to the element to be returned.
PDS **	ppds	A pointer to a PDS pointer.

Returns
On success PDS_ERR_NONE, and on error PDS_ERR. The specific error code is found with pdsError().

See also
pdsGet(), pdsScanf() and pdsCreate().

pdsGet_CPDS

Signature

```
int   pdsGet_CPDS(PDS *pds, char *key, PDS **ppPds);
```

Description
This convenience function will extract the PDS element named key (or the next element if key is NULL), and return a pointer to an allocated PDS structure provided by this API.
NOTE. Do not modify the contents of this data (unless you know what you're doing.).

Parameters

PDS *	pds	A pointer to an existing PDS.
char *	key	The key to the element to be returned.
PDS **	ppds	A pointer to an existing PDS into which the PDS element is opened.

Returns
On success PDS_ERR_NONE, and on error PDS_ERR. The specific error code is found with pdsError().

See also
pdsGet(), pdsScanf() and pdsCreate().

pdsGet_VOID

Signature

```
int pdsGet_VOID(PDS *pds, char *key, void **data, long *size);
```

Description
This convenience function will extract the PDS element named key (or the next element if key is NULL), and return a pointer to a allocated buffer provided by this API. The size of the data is returned in the address provided by the 'size' parameter.

Parameters

PDS *	pds	A pointer to an existing PDS.
char *	key	The key to the element to be returned.
void **	data	A pointer to an address, which will get the address of, the data returned.
long *	size	The size of the data returned.

Returns
On success PDS_ERR_NONE, and on error PDS_ERR. The specific error code is found with pdsError().

See also
pdsGet(), pdsScanf() and pdsCreate().

pdsGet_RGCH

Signature

```
int pdsGet_RGCH(PDS *pds, char *key, char *pch, long size);
```

Description
This convenience function will extract the PDS element named key (or the next element if key is NULL), and copy the contents into a preallocated buffer of length len, provided by the caller.

Parameters

PDS *	pds	A pointer to an existing PDS.
char *	key	The key to the element to be returned.
char *	pch	A pointer to an existing character buffer.
long	size	The size of the character buffer.

Returns
On success PDS_ERR_NONE, and on error PDS_ERR. The specific error code is found with pdsError().

See also
pdsGet(), pdsScanf() and pdsCreate().

pdsGet_CPCH

Signature

```
int pdsGet_CPCH(PDS *pds, char *key, char **ppch);
```

Description
This convenience function will extract the PDS element named key (or the next element if key is NULL), and return a pointer to the where the actual PDS buffer element is.
NOTE. Do not modify the contents of this data (unless you know what you're doing...)

Parameters

PDS *	pds	A pointer to an existing PDS.
char *	key	The key to the element to be returned.
char **	ppch	A pointer to the address of a character pointer.

Returns
On success PDS_ERR_NONE, and on error PDS_ERR. The specific error code is found with pdsError().

See also
pdsGet(), pdsScanf() and pdsCreate().

pdsGet_PCH

Signature

```
int pdsGet_PCH(PDS *pds, char *key, char **ppch);
```

Description
This convenience function will extract the PDS element named key (or the next element if key is NULL), and return a pointer to an allocated buffer provided by this API.

Parameters

PDS *	pds	A pointer to an existing PDS.
char *	key	The key to the element to be returned.
char **	ppch	A pointer to the address of a character pointer.

Returns
On success PDS_ERR_NONE, and on error PDS_ERR. The specific error code is found with pdsError().

See also
pdsGet(), pdsScanf() and pdsCreate().

pdsGet_PPCH

Signature

```
int pdsGet_PPCH(PDS *pds, char *key, char ***pppch);
```

Description
This convenience function will extract the PDS element named key (or the next element if key is NULL), and return a pointer to an allocated buffer provided by this API. This buffer is a null-terminated string table.

Parameters

PDS *	pds	A pointer to an existing PDS.
char *	key	The key to the element to be returned.
char ***	pppch	The pointer to an address which will get the string table pointer.

Returns
On success PDS_ERR_NONE, and on error PDS_ERR. The specific error code is found with pdsError().

See also
pdsGet(), pdsScanf() and pdsCreate()

pdsGet_INT

Signature

```
int pdsGet_INT(PDS *pds, char *key, int *pi);
```

Description
This convenience function will extract the PDS element named key (or the next element if key is NULL), and return the value of the integer in the address pointed to by data.

Parameters

PDS *	pds	A pointer to an existing PDS.
char *	key	The key to the element to be returned.
int *	pi	An address of an integer pointer.

Returns
On success PDS_ERR_NONE, and on error PDS_ERR. The specific error code is found with pdsError().

See also
pdsGet(), pdsScanf() and pdsCreate()

pdsGet_PI

Signature

```
int pdsGet_PI(PDS *pds, char *key, int **ppi);
```

Description
This convenience function will extract the PDS element named *key* (or the next element if *key* is NULL), and return a pointer to an allocated integer, in the address pointed to by *ppi*.

Parameters

PDS *	pds	A pointer to an existing PDS.
char *	key	The key to the element to be returned.
int **	ppi	A pointer to a pointer to an allocated integer.

Returns
On success PDS_ERR_NONE, and on error PDS_ERR. The specific error code is found with pdsError().

See also
pdsGet(), pdsScanf() and pdsCreate().

pdsGet_LONG

Signature

```
int pdsGet_LONG(PDS *pds, char *key, long *pl);
```

Description
This convenience function will extract the PDS element named *key* (or the next element if *key* is NULL), and return the value of the long integer in the address pointed to by *pl*.

Parameters

PDS *	pds	A pointer to an existing PDS.
char *	key	The key to the element to be returned.
long *	pl	A pointer to a long.

Returns
On success PDS_ERR_NONE, and on error PDS_ERR. The specific error code is found with pdsError().

See also
pdsGet(), pdsScanf() and pdsCreate()

pdsGet_PL

Signature

```
int pdsGet_PL(PDS *pds, char *key, long **ppl);
```

Description
This convenience function will extract the PDS element named key (or the next element if key is NULL), and return a pointer to an allocated long integer, in the address pointed to by data.

Parameters

PDS *	pds	A pointer to an existing PDS.
char *	key	The key to the element to be returned.
long **	ppl	The address of a pointer to a long.

Returns
On success PDS_ERR_NONE, and on error PDS_ERR. The specific error code is found with pdsError().

See also
pdsGet(), pdsScanf() and pdsCreate()

pdsGet_F

Signature

```
int pdsGet_F(PDS *pds, char *key, double *pd);
```

Description
This convenience function will extract the PDS element named *key* (or the next element if *key* is NULL), and return the value of the double in the address pointed to by *pd*.

Parameters

PDS *	pds	A pointer to an existing PDS.
char *	key	The key to the element to be returned.
double *	pd	The address of a double.

Returns
On success PDS_ERR_NONE, and on error PDS_ERR. The specific error code is found with pdsError().

See also
pdsGet(), pdsScanf() and pdsCreate()

pdsGet_SIZE

Signature

```
int pdsGet_SIZE(PDS *pds, char *key, size_t *ps);
```

Description
See description for other pdsGet functions.

Parameters

PDS	*pds	The PDS to read the value from
char	*key	The key that contains the size_t value we want to read.
size_t	*ps	The address of a size_t variable to fill in with the value

See also
pdsPut_SIZE

pdsGet_TIME

Signature

```
int pdsGet_TIME(PDS *pds, char *key, time_t *pt);
```

Description
See description for other pdsGet functions.

Parameters

PDS	*pds	The PDS to read the value from
char	*key	The key that contains the time_t value we want to read.
time_t	*pt	The address of a time_t variable to fill in with the value.

See also
pdsPut_TIME

pdsGetTable

Signature

```
int pdsGetTable(PDS *pds, PDStype t, char *key, int index, void *value);
```

Description
This function is the companion function of pdsPutTable(). It will extract table elements according to the index. The index is incremented on each run, by pdsGetTable. If idx is -1, it cleans the static temporary PDS.

Parameters

PDS *	pds	A pointer to an existing PDS.
PDStype	t	The type of elements in the table. (PDS_CPCH, PDS_PCH, PDS_INT, PDS_PDS, PDS_CPDS)
char *	key	The table name.
int	index	Which element to get. The first element has index 0.
void *	value	Pointer to an address, which will receive the table element.

Returns
On success: PDS_ERR_NONE, on error: PDS_ERR_ILLARG, PDS_ERR_ILLTYPE, PDS_ERR_NOMATCH or PDS_ERR_MALLOC.

See also
pdsPutTable()

pdsMap

Signature

```
PDS *pdsMap (PDS *pds, char *data, int size);
```

Description
This function will map an existing buffer into a PDS structure, and mark the data area as protected. If *pds* is NULL, a new PDS structure is allocated, and mapped with the buffer *data*.
If a *pds* parameter is given with a PDS, which has been created by pdsCreate(), then its data is freed before the mapping.
Note that the data is freed together with the PDS when pdsDelete() is called.

Parameters

PDS *	*pds*	Optional pointer to an existing PDS. If NULL is specified, a new PDS is created.
char*	data	The data section of the PDS is replaced with the given data.
init	size	Size of the data specified above.

Returns
Pointer to the updated PDS.

See also
pdsGet(), pdsCreate(), pdsPut() and pdsDelete()

pdsPrintf

Signature

```
int pdsPrintf(PDS *pds, char *format, ...);
```

Description
This function is equivalent to fprintf, in terms of how data is stored into a stream. The parameters passed on to the function should map on a one-to-one basis with the elements in the formatting string. The formatting specifiers are the same as the ones in pdsScanf().

Parameters

PDS *	pds	A pointer to an existing PDS.
char *	format	A text string describing the information you want to store. See below for a more detailed description.

Returns
PDS_ERR_NONE on success and PDS_ERR on failure. On failure, pdsError(pds) will give more information.

Example

```
int i;

char *name;

PDS *pds;

If (pds=pdsCreate())

   pdsPrintf(pds, “name,age%d”, name, i);
```

See also
pdsScanf(), pdsCreate()

pdsPutTable

Signature

```
int   pdsPutTable(PDS *pds,PDStype t, char *key, void *value);
```

Description
This function will generate indexed elements, from zero (0) and upwards. The table equivalent of the type is used as the target PDS. For instance will a type of PDS_PCH give a table of type PDS_PPCH.

Example

```
PDS *pds;

char *ppch;

pds = pdsCreate();

pdsPutTable(pds,PDS_PCH,”name”,”Ole”);

pdsPutTable(pds,PDS_PCH,”name”,”Dole”);

pdsPutTable(pds,PDS_PCH,”name”,”Doffen”);

pdsPutTable(pds,PDS_PCH,”name”,”Donald”);

pdsDump(pds);

pdsGet PPCH(pds,“names”,&ppch);
```

Parameters

PDS *	pds	A pointer to an existing PDS into which the elements are to be inserted.
PDStype	t	The type of table elements supplied. The supported types are PDS_PCH, PDS_INT and PDS_PDS.
char *	key	The table name.
void *	value	Pointer to an element to be added to the table.

Returns
On success: PDS_ERR_NONE, on error: PDS_ERR_ILLARG, PDS_ERR_ILLTYPE.

See also
pdsGet_TIME, odsGetTable, pdsDump.

pdsPut

Signature

```
int pdsPut(PDS *pds, PDStype t, char *key, void *d, long l);
```

Description
This function will put the data *d* of size *l* into the PDS stream pointed to by the supplied PDS handle. The data that is packed into the PDS stream, may be extracted by calling pdsGet() or any of the pdsGet_ macros defined in pds.h.
You may use the following types:

PDS_I - integer
PDS_PI - integer pointer
PDS_PPI - pointer to integer pointer
PDS_RGI - pointer to an existing integer
PDS_RGPI - pointer to an integer array
PDS_CH - character
PDS_PCH - 0-terminated string
PDS_PPCH - pointer to 0-terminated string
PDS_RGCH - pointer to a character buffer
PDS_RGPCH - pointer to a character array buffer
PDS_F - floating point number
PDS_PF - floating point number pointer
PDS_PPF - pointer to point number pointer
PDS_RGF - pointer to a floating point number
PDS_RGPF - pointer to a floating point number array
PDS_VOID - raw byte data
PDS_SEP - no data

The following convenience functions are available:

pdsPut_PCH (pds, pch)
pdsPut_PPCH (pds, ppch)
pdsPut_INT (pds, i)

Parameters

PDS *	pds	A pointer to an existing PDS into which to add the element.
PDStype	t	Type of element to be added
char *	key	The key to use to refer to the element.
void *	d	The element to be added.
long	l	The size of the element to be added.

Returns
0 on success and 1 on error. Use pdsError to get the actual error code.

See also
pdsGet(), pdsCreate() and pdsPut().

pdsPut_PDS

Signature

```
int pdsPut_PDS(PDS *pds, char *key, PDS *dpds);
```

Description
This convenience function will add a PDS element to an existing PDS. The element can later be referenced with the *key*.

Parameters

PDS *	pds	A pointer to an existing PDS.
char *	key	The key of the element to be added.
PDS *	dpds	A pointer to a PDS.

Returns
On success PDS_ERR_NONE, and on error PDS_ERR. The specific error code is found with pdsError().

See also
pdsCreate() and pdsPut()

pdsPut_PPDS

Signature

```
int pdsPut_PPDS(PDS *pds, char *key, PDS *dpds);
```

Description
This convenience function will add an encapsulated PDS table to an existing PDS. The element can later be referenced with the *key*.

Parameters

PDS *	pds	A pointer to an existing PDS.
char *	key	The key of the element to be added.
PDS *	dpds	The encapsulated PDS table to be added.

Returns
On success PDS_ERR_NONE, and on error PDS_ERR. The specific error code is found with pdsError().

See also
pdsCreate() and pdsPut()

pdsPut_VOID

Signature

```
int pdsPut_VOID(PDS *pds, char *key, void *d, long size);
```

Description
This convenience function will add byte data to an existing PDS. The data can later be referenced with the *key*.

Parameters

PDS *	pds	A pointer to an existing PDS.
char *	key	The key of the element to be added.
void *	d	A pointer to the byte data to be added.
long	size	The size of the data to be added.

Returns
On success PDS_ERR_NONE, and on error PDS_ERR. The specific error code is found with pdsError().

See also
pdsCreate() and pdsPut()

pdsPut_RGCH

Signature

```
int pdsPut_RGCH(PDS *pds, char *key, char *pch, long size);
```

Description
This convenience function will add a string to an existing PDS. The string can later be referenced with the *key*.

Parameters

PDS *	pds	A pointer to an existing PDS.
char *	key	The key of the element to be added.
char *	pch	The string to be added.
long	size	The size of the string to be added.

Returns
On success PDS_ERR_NONE, and on error PDS_ERR. The specific error code is found with pdsError().

See also
pdsCreate() and pdsPut()

pdsPut_PCH

Signature

```
int pdsPut_PCH(PDS *pds, char *key, char *pch);
```

Description
This convenience function will add a 0-terminated string to an existing PDS. The string can later be referenced with the *key*.

Parameters

PDS *	pds	A pointer to an existing PDS.
char *	key	The key of the element to be added.
char *	pch	The 0-terminated string to be added.

Returns
On success PDS_ERR_NONE, and on error PDS_ERR. The specific error code is found with pdsError().

See also
pdsCreate() and pdsPut()

pdsPut_PPCH

Signature

```
int pdsPut_PPCH(PDS *pds, char *key, char **ppch);
```

Description
This convenience function will add a string table to an existing PDS. The string table can later be referenced with the *key*.

Parameters

PDS *	pds	A pointer to an existing PDS.
char *	key	The key of the element to be added.
char **	ppch	The string table to be added.

Returns
On success PDS_ERR_NONE, and on error PDS_ERR. The specific error code is found with pdsError().

See also
pdsCreate, pdsPutTable

pdsPut_INT

Signature

```
int pdsPut_INT(PDS *pds, char *key, int i);
```

Description
This convenience function will add an integer to an existing PDS. The integer can later be referenced with the key.

Parameters

PDS *	pds	A pointer to an existing PDS.
char *	key	The key of the element to be added.
int	i	The integer to be added.

Returns
On success PDS_ERR_NONE, and on error PDS_ERR. The specific error code is found with pdsError().

See also
pdsCreate() and pdsPut()

pdsPut_PPI

Signature

```
int pdsPut_PPI(PDS *pds, char *key, int *pi);
```

Description
This convenience function will add an integer table to an existing PDS. The integer table can later be referenced with the *key*.

Parameters

PDS *	*pds*	A pointer to an existing PDS.
char *	key	The key of the element to be added.
int *	pi	The integer table to be added.

Returns
On success PDS_ERR_NONE, and on error PDS_ERR. The specific error code is found with pdsError().

See also
pdsCreate, pdsPutTable

pdsPut_LONG

Signature
int pdsPut_LONG(PDS *pds, char *key, long l);

Description
This convenience function will add a long to an existing PDS. The long can later be referenced with the *key*.

Parameters

PDS *	pds	A pointer to an existing PDS.
char *	key	The key of the element to be added.
long	l	The long to be added.

Returns
On success PDS_ERR_NONE, and on error PDS_ERR. The specific error code is found with pdsError().

See also
pdsCreate() and pdsPut()

pdsPut_PPL

Signature

```
int pdsPut_PPL(PDS *pds, char *key, long *pl);
```

Description
This convenience function will add a long table to an existing PDS. The long table can later be referenced with the *key*.

Parameters

PDS *	pds	A pointer to an existing PDS.
char *	key	The key of the element to be added.
long *	pl	The long table to be added.

Returns
On success PDS_ERR_NONE, and on error PDS_ERR. The specific error code is found with pdsError().

See also
pdsCreate, pdsPutTable

pdsPut_F

Signature

```
int pdsPut_F(PDS *pds, char *key, double f);
```

Description
This convenience function will add a double to an existing PDS. The double can later be referenced with the *key*.

Parameters

PDS *	*pds*	A pointer to an existing PDS.
char *	key	The key of the element to be added.
double	f	The double to be added.

Returns
On success PDS_ERR_NONE, and on error PDS_ERR. The specific error code is found with pdsError().

See also
pdsCreate() and pdsPut()

Put_SIZE

Signature

```
Int pdsPut_SIZE(PDS *pds, char *key, size_t s);
```

Description
See the descriptions for other pdsPut functions.

Parameters

PDS	*pds	The PDS to insert the new key/value pair into.
char	*key	The name of the key.
size_t	t	The size_t value to insert into the PDS.

Put_TIME

Signature

```
Int pdsPut_TIME(PDS *pds, char *key, time_t t);
```

Description
See description for other pdsPut functions.

Parameters

PDS	*pds	The PDS to insert the new key/value pair into.
char	*key	The name of the key.
time_t	t	The time_t value to insert into the PDS

pdsRemove

Signature

```
int pdsRemove(PDS *pds, char *key);
```

Description
This function will remove the element named *key* from the PDS stream.

Parameters

PDS *	pds	A pointer to an existing PDS.
Char *	key	The name of a key existing in the given PDS.

Returns
PDS_ERR_NONE if the element is successfully removed, and PDS_ERR if it is not found.

See also
pdsCreate() and pdsSet()

pdsRewind

Signature

```
int pdsRewind(PDS *pds);
```

Description
This function will reinitialize the get pointer within the PDS object. This function is used when you want to perform multiple gets on the PDS.

Parameters

PDS *	pds	A pointer to an existing PDS.

Returns
PDS_ERR_NONE on success, PDS_ERR if the PDS does not exist.

See also
pdsCreate(), pdsPut(), pdsGet() and pdsReset()

pdsReset

Signature

```
int pdsReset(PDS *pds);
```

Description
This function will reinitialize the PDS object, to an empty buffer, and initial pointer settings. If you wish to reset the get pointer please use the pdsRewind() function. This function will wipe out your data. The buffer is not reallocated.

Parameters

PDS *	pds	A pointer to an existing PDS.

Returns
PDS_ERR_NONE on success, PDS_ERR if the PDS does not exist.

See also
pdsCreate(), pdsPut(), pdsGet() and pdsRewind()

pdsSet

Signature

```
PDS *pdsSet(char *data, int size);
```

Description
This function will create a new PDS stream just like pdsCreate(), but it will set the provided data as the data

stream. It is useful when a stream has been stored in for instance a database and extracted for dissection. Note that a new memory area is allocated, into which the data specified is copied.

Parameters

char *	data	The data, which will be set in the PDS that was created.
int	size	The size of the above data.

Returns
A pointer to a new PDS, or NULL.

See also
pdsCreate(), pdsPut(), pdsGet() and pdsDelete()

pdsScanf

Signature

```
int pdsScanf(PDS *pds, char *format, ...);
```

Description
This function will extract (and convert) data from the provided PDS, using the format string. In other words, you may extract a PDS_PCH as an integer, long or double using the formatting codes. You may only extract / convert the following types: PDS_PCH, PDS_RGCH, PDS_I, and PDS_F.

The following formatting codes are supported:

%s	string	stored in char **
%b	string	copied to buffer pointed to by char *
%d	decimal	stored in int *
%l	decimal	stored in long *
%f	float	stored in double *
%T	PDS data type	stored in int *
%L	PDS data length	stored in int *

In addition to the formatting specifier, you may indicate the length of the storage area as done with scanf().

Example

```
PDS *pds;
int iAge;
char *pchName, szAge[20];

pds = pdsCreate();
pdsPut_PCH(pds, “name”, “Test user”);
pdsPut INT(pds, “age”, 35);

pdsScanf(pds, “name,age%10b,age%d”, pchName, szAge, &iAge);
pdsDelete(pds);
```

Parameters

PDS *	pds	A pointer to an existing PDS from which you want to extract information.
char *	format	A text string describing the information you want to extract. See below for a more detailed description.

Returns
PDS_ERR_NONE on success and PDS_ERR on failure. On failure pdsError(pds) will give more information.

See also
pdsPrintf(), pdsGet()

CFG - The configuration library

cfgClose

Signature

```
int cfgClose(cfgHandle *cfgh);
```

Description
If the contents of the configuration have changed then these changes are written back to the configuration file, if it was no opened as read-only. Memory allocated to the cfgHandle structure is freed.

Parameters

cfgHandle *	*cfgh*	A pointer to a structure containing the configuration data.

Returns
TRUE on success

FALSE if an error occurred.

See also
cfgOpen

cfgOpen

Signature

```
cfgHandle * cfgOpen(char *pchFile,int bReadOnly);
```

Description
cfgOpen opens a configuration file and reads the contents of that file into a cfgHandle structure. If bReadOnly is FALSE the file is opened in read-write mode. If the user does not have read-write access to the call will fail. If the user does not have read access to the file, or the file does not exist, the call will fail.

The pointer to the cfgHandle structure, which is returned, is used by the rest of the configuration handling functions.

Parameters

char *	pchFile	String containing the name of the file that should be opened. The path will be relative to the program location unless a full path is specified.
Int	bReadOnly	Opens the file in read mode if TRUE, in read-write mode if FALSE.

Returns
A pointer to a cfgHandle structure is returned if the file is read successfully. A NULL pointer is returned if an error occurs.

See also
cfgClose

cfgKeyWrite

Signature

```
int   cfgKeyWrite(cfgHandle *cfgh, char *sec, char *key, char *val);
```

Description
cfgKeyWrite will insert or change a value in a section. If the section does not exist it is created. If the key exists the value is changed otherwise the key and value are added to the section. If value (val) is NULL the key is created with out a value. If key is NULL only the section is created.

Parameters

cfgHandle*	cfgh	A pointer to a structure containing the configuration data.
char *	sec	The section name that the key belongs to.

char *	key	The name of the key.
char *	*val*	The value of the key.

Returns
TRUE on success
FALSE if an error occurred.

See also
cfgOpen, cfgKeyRead, cfgKeyWriteInt, cfgKeyRename, cfgKeyWriteDouble

cfgKeyWriteInt

Signature

```
int   cfgKeyWriteInt(cfgHandle *cfgh,char *sec,char *key, int val)
```

Description
cfgKeyWriteInt is a conveniance function on top of cfgKeyWrite. Since cfgKeyWrite takes a char * as the value a conversion routine was required. This function does the conversion for you and then calls cfgKeyWrite.

Parameters

cfgHandle*	cfgh	A pointer to a structure containing the configuration data.
char *	sec	The section name that the key belongs to.
char *	key	The name of the key.
Int	val	The value of the key.

Returns
TRUE on success
FALSE if an error occurred.

See also
cfgOpen, cfgKeyRead, cfgKeyWrite, cfgKeyWriteLong, cfgKeyWriteDouble

cfgKeyWriteLong

Signature

```
int   cfgKeyWriteLong(cfgHandle *cfgh,char *sec,char *key, long val);
```

Description
cfgKeyWriteLong is a conveniance function on top of cfgKeyWrite. Since cfgKeyWrite takes a char * as the value a conversion routine was required. This function does the conversion for you and then calls cfgKeyWrite.

Parameters

cfgHandle*	cfgh	A pointer to a structure containing the configuration data.
char *	sec	The section name that the key belongs to.
char *	key	The name of the key.
long	val	The value of the key.

Returns
TRUE on success
FALSE if an error occurred.

See also
cfgOpen, cfgKeyRead, cfgKeyWrite, cfgKeyWriteInt, cfgKeyWriteDouble

cfgKeyWriteDouble

Signature

```
int   cfgKeyWriteDouble(cfgHandle *cfgh,char *sec,char *key, double val);
```

Description
cfgKeyWriteLong is a conveniance function on top of cfgKeyWrite. Since cfgKeyWrite takes a char * as the value a conversion routine was required. This function does the conversion for you and then calls cfgKeyWrite.

Parameters

cfgHandle*	cfgh	A pointer to a structure containing the configuration data.
char *	sec	The section name that the key belongs to.
char *	key	The name of the key.
double	val	The value of the key.

Returns
TRUE on success
FALSE if an error occurred.

See also
cfgOpen, cfgKeyRead, cfgKeyWrite, cfgKeyWriteInt, cfgKeyWriteLong

cfgKeyRead

Signature

```
char *      cfgKeyRead(cfgHandle *cfgh, char *sec, char *key);
```

Description
cfgKeyRead returns an allocated string with the value of the key. Returns NULL if the key or the section does not exist. The user must free the string after use.

Parameters

cfgHandle*	cfgh	A pointer to a structure containing the configuration data.
char *	sec	The section name that the key belongs to.
char *	key	The name of the key.

Returns
A pointer to an allocated string is returned on success.
A NULL pointer is returned on error.

See also
cfgOpen, cfgKeyWrite, cfgKeyReadInt, cfgKeyReadLong, cfgKeyReadDouble, cfgSectionExist.

cfgKeyReadStr

Signature

```
char *      cfgKeyReadStr(cfgHandle *cfgh, char *sec, char *key, char *def);
```

Description
Wrapper function for cfgKeyRead which saves you the trouble of doing the conversioin from char * to double yourself. Specify the value 'def' which is returned if the key cannot be found.

Parameters

cfgHandle*	cfgh	A pointer to a structure containing the configuration data returned by cfgOpen.
char *	sec	The section name that the key belongs to.
char *	key	The name of the key to read.
char *	def	The default value to use if key does not exist or is blank.

Returns
String with the value of the variable *key* or *def* if key does not exist or is blank. NULL if *def* is not set.

See also
cfgOpen, cfgKeyWrite, cfgKeyReadInt, cfgKeyReadLong, cfgKeyReadDouble, cfgKeyRead.

cfgKeyReadInt

Signature

```
int   cfgKeyReadInt(cfgHandle *cfgh, char *sec, char *key, int def);
```

Description
cfgKeyReadInt is a convenience function on top of cfgKeyRead. Since cfgKeyRead returns an allocated string conversion needed to be done. This function does the necessary conversion for you and frees the memory allocated in cfgKeyRead automatically.

Parameters

cfgHandle*	cfgh	A pointer to a structure containing the configuration data.
char *	sec	The section name that the key belongs to.
char *	key	The name of the key.
int	def	The default value to assing if reading the value of key fails.

Returns
The value of the key as an integer on success, or the default value def.

See also
cfgOpen, cfgKeyWrite, cfgKeyRead, cfgKeyReadLong, cfgKeyReadDouble

cfgKeyReadLong

Signature

```
long  cfgKeyReadLong(cfgHandle *cfgh, char *sec, char *key, long def);
```

Description
cfgKeyReadLong is a convenience function on top of cfgKeyRead. Since cfgKeyRead returns an allocated string conversion needed to be done. This function does the necessary conversion for you and frees the memory allocated in cfgKeyRead automatically.

Parameters

cfgHandle*	cfgh	A pointer to a structure containing the configuration data.
char *	sec	The section name that the key belongs to.
char *	key	The name of the key.
long	def	The default value to assing if reading the value of key fails.

Returns
The value of key as a long on success, or the default value def.

See also
cfgOpen, cfgKeyWrite, cfgKeyRead, cfgKeyReadInt, cfgKeyReadDouble

cfgKeyReadDouble

Signature

```
double      cfgKeyReadDouble(cfgHandle *cfgh, char *sec, char *key, double def);
```

Description
cfgKeyReadDouble is a convenience function on top of cfgKeyRead. Since cfgKeyRead returns an allocated string conversion needed to be done. This function does the necessary conversion for you and frees the memory allocated in cfgKeyRead automatically.

Parameters

cfgHandle*	cfgh	A pointer to a structure containing the configuration data.
char *	sec	The section name that the key belongs to.
char *	key	The name of the key.
double	def	The default value to assing if reading the value of key fails.

Returns
The value of the key as a double on success, or the default value def.

See also
cfgOpen, cfgKeyWrite, cfgKeyRead, cfgKeyReadInt, cfgKeyReadLong.

cfgKeyReadYesNo

Signature

```
Int cfgKeyReadYesNo(cfgHandle *cfgh, char *sec, char *key, int def);
```

Description
cfgKeyReadYesNo is a convenience function on top of cfgKeyRead. Since cfgKeyRead returns an allocated string conversion into an integer value (0 or 1) needs to be done. The function will return true (1) if the value is one of the following: yes, true, on, enable, 1. Any other value will return 0. If the function fails to read a value for the given key then the default value is returned.

Parameters

cfgHandle	*cfgh	A pointer to a structure containing the configuration data.
char	*sec	The section name that the key belongs to.
char	*key	The name of the key.
int	def	The default value to assign if reading the value of the key fails.

cfgKeyDelete

Signature

```
int cgKeyDelete(cfgHandle *cfgh, char *sec, char *key);
```

Description
cfgKeyDelete will delete a key and its value from a section. The key and the section must exist.

Parameters

cfgHandle*	cfgh	A pointer to a structure containing the configuration data.
char *	sec	The section name that the key belongs to.
char *	key	The name of the key.

Returns
TRUE on success.
FALSE if an error occurred.

See also
cfgOpen, cfgKeyRename, cfgSectionExist.

cfgKeyRename

Signature

```
int   cfgKeyRename(cfgHandle *cfgh, char *sec, char *old_key, char *new_key);
```

Description
cfgKeyRename will change the name of an existing key, keeping its value unchanged. The section and old_key must exist, and new_key must not exist. The name of the section cannot be changed with this function. If you wish to change the name of the section as well as the name of the key then you must use cfgKeyWrite to create the new section and key, and cfgKeyDelete to remove the existing key.

Parameters

cfgHandle*	cfgh	A pointer to a structure containing the configuration data.
char *	sec	The section name that the key belongs to.
char *	old_key	The old name of the key.
char *	new_key	The new name of the key.

Returns
TRUE on success.
FALSE if an error occurred.

See also
cfgOpen, cfgKeyWrite, cfgKeyReadYesNo

cfgKeyList

Signature

```
char **     cfgKeyList(cfgHandle *cfgh, char *sec);
```

Description
cfgKeyList returns a list of keys found in the specified section as a string table. You can loop through the string table until you encounter a NULL pointer. It is up to the caller to free the string table after use (both the contents of the table and the table itself must be freed).

Parameters

cfgHandle*	cfgh	A pointer to a structure containing the configuration data.
char *	sec	The name of the section to list all keys in.

Returns
An allocated string table containing the keys found in the section sec.

See also
cfgOpen, cfgKeyRead, cfgSectionExist, cslTblFree.

cfgListWrite

Signature

```
int   cfgListWrite(cfgHandle *cfgh, char *sec, char *key, char **list);
```

Description
cfgListWrite will replace a section with the contents of the stringtable list. Each entry in the string table will be treated as a value. The key names are generated in such a way that if e.g key is “file” then the first entry in the string table will have the key “file_0”, the second entry will have the key “file_1” and so on.

Parameters

cfgHandle*	cfgh	A pointer to a structure containing the configuration data.
char *	sec	The name of the section to add the table values to
char *	key	The basis name for the keys that are generated.
char **	list	The string table containing values that are to be added.

Returns
TRUE on success.
FALSE if an error occurred.

See also
cfgOpen, cfgListRead, cfgKeyRead, cfgSectionExist.

cfgListRead

Signature

```
char **     cfgListRead(cfgHandle *cfgh, char *sec);
```

Description
cfgListRead reads the section specified and returns the list of values as a string table. The user must free the string table after use, both the values and the table itself.

Parameters

cfgHandle*	cfgh	A pointer to a structure containing the configuration data.
char *	sec	The name of the section to add the table values to.

Returns
An allocated string table containing the values found in the section sec. A NULL pointer is returned if the section does not exist or is empty.

See also
cfgOpen, cfgListWrite, cslTblFree.

cfgSaveAs

Signature

```
int   cfgSaveAs(cfgHandle *cfgh,const char *newfile);
```

Description
cfgSaveAs will save the contents of the configuration to a new file, storing the new file name in the cfgHandle structure. The old filename is discarded.

Parameters

cfgHandle*	cfgh	A pointer to a structure containing the configuration data.
const char*	newfile	The name of the file that the configuration data should be saved as.

Returns
TRUE on success.
FALSE if an error occurred.

See also
cfgClose, CfgSectionMoveLast

CfgSectionMoveLast

Signature

```
int cfgSectionMoveLast(cfgHandle *cfgh, char *sec);
```

Description
cfgSectionMoveLast will move the specified section to the end of the configuration. Note that the cfgh handle must have been opened with write permissions.
Parameters

cfgHandle*	cfgh	A pointer to a structure containing the configuration data.
char *	sec	The name of the section to move within the configuration data.

Note
There are several functions in the header file that begin with _cfg and which are internal functions not for use outside the library.

cfgSectionDelete

Signature

```
int cfgSectionDelete(cfgHandle *cfgh, char *sec);
```

Description
Deletes a section in the configuration file. Returns an error if the section does not exist.

Parameters

cfgHandle*	cfgh	A pointer to a structure containing the configuration data.
char *	sec	The name of the section to delete.

Returns
TRUE on success
FALSE if an error occurred.

See also
cfgOpen, cfgKeyReadYesNo

cfgSectionRename

Signature

```
int   cfgSectionRename(cfgHandle *cfgh, char *from_sec, char *to_sec);
```

Description
cfgSectionRename will change the name of a section. It will return an error if the section does not exist. You must specify the full section path in the from_sec parameter, but only the section name in the to_sec. NOTE: You can get two sections with the same name if you change to a section name that already exists at the same level.

Parameters

cfgHandle*	cfgh	A pointer to a structure containing the configuration data.
char *	from_sec	The old name of the section.
char *	to_sec	The new name of the section.

Returns
TRUE on success
FALSE if an error occurred.

See also
cfgOpen, cfgSectionDelete, cfgSectionCopy, cfgSectionExist.

cfgSectionCopy

Signature

```
int   cfgSectionCopy(cfgHandle *cfgh, char *from_sec, char *to_sec);
```

Description
cfgSectionCopy will copy a section to a new name at the same level. You must specify the full section path in the from_sec, but should only use the section name in to_sec. NOTE: You can get two sections with the same name if you copy to a section name that already exists at the same level.

Parameters

cfgHandle*	cfgh	A pointer to a structure containing the configuration data.
char *	from_sec	The old name of the section.
char *	to_sec	The new name of the section.

Returns
TRUE on success.
FALSE if an error occurred.

See also
cfgOpen, cfgSectionDelete, cfgSectionRename, cfgSectionExist

cfgSectionExist

Signature

```
int   cfgSectionExist(cfgHandle *cfgh, char *sec);
```

Description
cfgSectionExist checks that the section exists.

Parameters

CfgHandle*	cfgh	A pointer to a structure containing the configuration data.
char *	sec	The name of the section to delete.

Returns
TRUE if section sec exists
FALSE if the section does not exist

See also
cfgKeyRead, cfgListRead, cfgSectionDelete, cfgSectionRename, cfgSectionCopy

cfgSectionList

Signature

```
char **     cfgSectionList(cfgHandle *cfgh, char *in_sec, int bRecurse);
```

Description
cfgSectionList creates a list of all sections in a file if 'sec' is NULL. If a 'sec' is given only sections within that section is listed. The sections are given with full section path. If bRecurse is TRUE then all sections contained in sections below the current section are included in the string table as well. The caller must free the string table (both the strings it contains and the table itself).

Parameters

cfgHandle*	cfgh	A pointer to a structure containing the configuration data.
char *	in_sec	The name of the section to begin searching in.
int	bRecurse	Flags if the list should recurse to sections found within the current section.

Returns
An allocated string table of sections on success. A NULL pointer if an error has occurred.

See also
cfgOpen, cfgKeyList, cslTblFree

cfgSectionMoveLast

Signature

```
Int cfgSectionMoveLast(cfgHandle *cfgh, char *sec);
```

Description

cfgSectionMoveLast will move the specified section to the end of the configuration. Note that the cfgh handle must have been opened with write permissions.

Note: There are several functions in the header file that begin with _cfg and which are internal functions not for use outside the library.

Parameters

cfgHandle	*cfgh	A pointer to a structure containing the configuration data.
char	*sec	The name of the section to move within the configuration data.

cfgSync

Signature

```
int   cfgSync(cfgHandle *cfgh);
```

Description
cfgSync will write all changes in the configuration to the open file without closing the handle.

Parameters

cfgHandle*	*cfgh*	A pointer to a structure containing the configuration data.

Returns
TRUE on success.
FALSE if an error occurred.

See also
cfgOpen, cfgClose, cfgSaveAs

CSL - The common string library

cslConvertBase

Signature

```
Char *cslConvertBase(long number, int base);
```

Description
Convert a number to a different base, like 16 (hexadecimal) or 8 (octal).

Parameters

long	number	Number to convert
int	base	Base to convert to (e.g 16=HEX)

cslFileWrite

Signature

```
int   cslFileWrite(FILE *fp, char **ppch);
```

Description
cslFileWrite will write the contents of the string table ppch to file. Line shifts are added to the lines if they are not already present.

Parameters

FILE *	Fp	Pointer to a file that has been opened for writing.
char **	ppch	A NULL terminated string table which is to be written to the file.

Returns
Number of lines written or 0 if a failure has occurred.

See also
cslFileRead

cslFileRead

Signature

```
char **     cslFileRead(FILE *fp);
```

Description
cslFileRead returns a NULL terminated string table created from an open ASCII file. Line shift is stripped from each line. The calling function must free the string table.

Parameters

FILE *	*fp*	Pointer to a file that has been opened for writing.

Returns
An allocated string table containing the contents of the file is returned on success.
A NULL pointer is returned and errno is set if an error occurred.

See also
cslFileWrite, cslTblFree

cslLineDelete

Signature

```
char **     cslLineDelete(char **ppch, int pos);
```

Description
cslLineDelete removes the line at offset *pos* from the string table. The rest of the strings are moved to fill the hole in the table.

Parameters

char **	ppch	A NULL terminated string table from which a line should be removed.
int	pos	Offset in the string table of the line to be deleted

Returns
The ppch with the line at offset *pos* removed. A NULL pointer is returned on error.

See also
cslLineInsert, cslLineReplace, cslTblFree

cslLineInsert

Signature

```
char **     cslLineInsert(char **ppch, char *line, int pos);
```

Description
cslLineInsert adds a line to the string table ppch at the given position. The position 0 is the start of the table, and the position –1 is the end of the table. All necessary memory is allocated by the function, and must be freed by the calling function.

Parameters

char **	ppch	A NULL terminated string table from which a line should be removed.
char *	line	String to be inserted into the table.
int	pos	Offset in the string table of the line to be inserted.

Returns
The ppch with the new line inserted at offset *pos*. A NULL pointer is returned on error.

See also
cslLineDelete, cslLineReplace, cslTblFree

cslLineReplace

Signature

```
char **     cslLineReplace(char **ppch, char *line, int pos);
```

Description
cslLineReplace will replace the existing string at offset *pos* in the *ppch* string table with the new *line*. The position 0 is the start of the table, and the position –1 is the end of the table. All necessary memory is allocated by the function, and must be freed by the calling function.

Parameters

char **	ppch	A NULL terminated string table from which a line should be removed.
char *	line	String to replace what is in the table at offset pos.
int	pos	Offset in the string table of the line to be replaced.

Returns
The ppch with the new line replacing the old one at offset *pos*. A NULL pointer is returned on error.

See also
cslLineInsert, cslLineDelete, cslTblFree

cslTblFree

Signature

```
char **     cslTblFree(char **ppch);
```

Description
cslTblFree will free the memory allocated to a NULL terminated string table. Both the memory used by the strings and the table itself are freed.

Parameters

char	*ppch*	A NULL terminated string table from which a line should be removed.

Returns
A NULL pointer.

See also
cslFileRead, cslLineInsert, cslLineDelete, cfgSectionList, cfgKeyList, cfgListRead

cslTblToStr

Signature

```
Char *cslTblToStr(char **a, char *eol);
```

Description
Converts a string table into a string, optionally inserting an end-of-line string after each entry. The string grows by 1k blocks to fit the text in the string table. This function allocates memory which it is the callers responsibility to free.

Parameters

char	**a	An array of strings to concatenate into a single string.
char	*eol	An optional end-of-line string to insert between elements of the array

See also

cslStrTokTable, cslLineInsert, cslFileRead

cslPatternToRegExp

Signature

```
void  cslPatternToRegExp(const char *pattern, char *reg_exp);
```

Description
cslPatternToRegExp will build a regular expression from pattern and place it into the buffer pointed to by regexp. Make sure that the buffer contains enough space.

Parameters

char *	pattern	A string to be compiled into a regular expression
char *	reg_exp	Buffer in which the compiled regexp is stored.

Returns
Nothing, the regular expression is placed in the buffer *reg_exp*.

See also
cslMatchRegExp, cslRegExpCompile

cslRegExpCompile

Signature

```
void *      cslRegExpCompile (const char *reg_exp);
```

Description
cslRegExpCompile will compile the string representation of the regular expression into a format suitable for use in the cslRegExpMatch function. Please note that care must be taken not to feed the compiled regular expression to cslMatchRegExp, as this takes an uncompiled regexp and compiles it before attempting to match it. If the input can be a pattern as well as a full regular expression the cslPatternToRegExp function should be called first, to ensure that the string fed to cslRegExpCompile is a valid regular expression.

Parameters

const char *	*reg_exp*	A string containing a regular expression that is compiled into a native representation for the regexp matching engine.

Returns
A compiled regular expression suitable for passing directly to the regular expression matching engine.

See also
cslTblToStr, cslPatternToRegExp, cslRegExpMatch

cslRegExpExec

Signature

```
int cslRegExpExec (const char *str, void *pattern, int *offsets, int offsetcount);
```

Description
Check a regular expression against a string and extract the beginning and end of any grouped matches into the offsets array. The size of the array must be ((1 + maximum number of matches supported) *3), so an array size (offsetcount) of 99 will support up to 32 matches.

The function returns 1+number_of_matches on successful match and -1 on failure to match the regex. The offsets start at offsets[match_number * 2] and end at offsets[(match_number * 2) +1]. Thus the first match ($0 in many parsing tools) begins at offset[0] and ends at offset[1] into the string, the second match ($1) is found between offset[1] and offset[2].

This is a highly specialized function, to be used to extract parts of a string using regular expressions. Normal usage in most probes would be to simply see if a string matches, and for that you should see the cslRegExpMatch and cslMatchRegExp functions!

Parameters

const char	*str	String to match regexp against.
void	*pattern	Regular Expression pattern to match.
int	*offsets	Array to keep offsets where matches are found.
int	offsetcount	Number of elements in the offsets array .

cslRegExpMatch

Signature

```
int   cslRegExpMatch (const char *string, void *pattern);
```

Description
cslRegExpMatch will check the contents of *string* against the compiled regular expression in *pattern*. If the regexp matches then TRUE is returned. This function should be used if the same regular expression is to be used several times, as the overhead of compiling the regexp is incurred only once (at the price of having to store the compiled regexp in memory).

Parameters

const char *	string	The string that the compiled regexp *pattern* is matched against.
void *	pattern	A compiled regular expression.

Returns
TRUE if the compiled regexp in *pattern* matches the contents of *string*.
FALSE otherwise.

See also
cslTblToStr, cslPatternToRegExp, cslRegExpCompile.

cslMatchRegExp

Signature

```
int   cslMatchRegExp (const char *str, const char *reg_exp);
```

Description
cslMatchRegExp is a convenience routine that will process the string in *reg_exp*, compile the resulting regexp and attempt to match it against *str*. This routine must not be confused with cslRegExpMatch, which takes a compiled regular expression as a parameter! This function can be used if the overhead of having to compile the regular expression every time a match is attempted is acceptable.

Parameters

const char *	str	The string that the pattern or regular expression in reg_exp is matched against.
void *	reg_exp	A pattern or full regular expression.

Returns
TRUE if the string contains a match for *reg_exp*. FALSE otherwise.

See also
cslTblToStr, cslRegExpCompile, cslRegExpMatch

cslYesOrNo

Signature

```
int   cslYesOrNo(char *str,int freeMem);
```

Description
cslYesOrNo will check the string *str* and see if it contains either “yes”, “enable” or “1”. The flag *freeMem* specifies if the string should be freed by this function.

Parameters

char *	str	The string that is to be evaluated as containing Yes or No.
int	freeMem	A flag that specifies if the string should be freed. TRUE will free the memory in *str*.

Returns
TRUE if the string contains Yes, Enable or 1 (in any CaSe)
FALSE otherwise.

See also
cfgKeyRead

cslStrToSec

Signature

```
int   cslStrToSec (char *str);
```

Description
cslStrToSec will evaluate the time specification as follows and convert it to seconds. Only the first letter of the spec is evaluated.

number[ ]<spec> [[,] number[ ]<spec>]
where spec is: s,sec,seconds,secundo.. - seconds
m,min,minutes,minuti... - minutes
h,hrs,hour(s),.. - hours
d,day,days,.. - days

Example
int s = cslStrToSec("10d,2hours,5min");

Parameters

char *	*str*	The time specification that is to be evaluated.

Returns
Number of seconds, or 0 on error.

cslStrExpand

Signature

```
char  *     cslStrExpand (const char *cpchIn, char *(*Func)(const char *));
```

Description
cslStrToSec will evaluate the time specification as follows and convert it to seconds. Only the first letter of the spec is evaluated.

number[ ]<spec> [[,] number[ ]<spec>]
where spec is: s,sec,seconds,secundo.. - seconds
m,min,minutes,minuti... - minutes
h,hrs,hour(s),.. - hour
d,day,days,.. - days

Example

```
int s = cslStrToSec("10d,2hours,5min");
```

Parameters

const char *	cpchIn	
char *	freeMem	

Returns
Number of seconds, or 0 on error.

cslStrExpandEx

Signature

```
char *      cslStrExpandEx (const char *cpchIn, char *(*Func)(const char *,const void *),void *pData);
```

Description
cslStrExpandEx takes the parameter cpchIn and uses the function specified in the second parameter to expand the variables. Variables begin with '$'. If a data block is needed to expand variables from (typically a PDS), then the void *pData should be used to send the data into the function. If the function Func returns a NULL if a variable cannot be expanded then the variable is re-inserted into the output string, otherwise the returned string is used (even if it is empty)

Parameters

const char *	cpchIn	The string that is to be expanded.
char *	freeMem	Pointer to a function that takes a char * and a void * and returns a char *.
void *	pData	The data structure which is used to expand the variables.

Returns
An allocated copy of string cpchIn with all $variables expanded. If a variable cannot be expanded it is up to the function specified whether the variable should be left in place or replaced with a blank space.

See also
cslStrExpandEnv.

cslStrExpandEnv

Signature

```
char *      cslStrExpandEnv (const char *cpchIn);
```

Description
cslStrExpandEnv takes the parameter cpchIn and uses the getenv system call to expand variables. Variables start with a '$'. If a variable cannot be expanded it is replaced with an empty string.

Parameters

const char *	*cpchIn*	The string that is to be expanded.

Returns
An allocated copy of the string cpchIn with all $variables expanded using the getenv function call.

See also
cslStrExpandEx

cslStrTokTable

Signature

```
char **     cslStrTokTable (char *str,int size,char *token);
```

Description
cslStrTokTable takes a string (str) and splits it up into pieces based on which tokens are defined. To split a sentence into its separate words you would use a token " " (space) and get a string table with each word as an element of that table returned. Since the returned string table is allocated the calling function must the memory.

Parameters

char *	str	The string that is to be split into a string table.
int	size	The length of the string in the first parameter.
char *	token	The tokens used to split the string.

Returns
An allocated string table of the input string having been split up where tokens were encountered or NULL if an error occurred.

See also
cslTblFree

cslToUpper

Signature

```
char *      cslToUpper(char *str);
```

Description
cslToUpper changes a string to all uppercase letters. It changes the string in place, so if the original string is required then a copy must be made before calling cslToUpper.

Parameters

char *	*str*	The string that is to be changed to uppercase.

Returns
A pointer to str after it has been changed to uppercase letters, or NULL if an error has occurred.

See also
cslToLower

cslToLower

Signature

```
char *      cslToLower(char *str);
```

Description
cslToLower changes a string to all lowercase letters. It changes the string in place, so if the original string is required then a copy must be made before calling cslToLower.

Parameters

char *	*str*	The string that is to be changed to lowercase.

Returns
A pointer to str after it has been changed to lowercase letters, or NULL if an error has occurred.

See also
cslToUpper

cslIsNumeric

Signature

```
char *      cslIsNumeric(char *str);
```

Description
cslIsNumeric checks if a given string is a numeric value. It supports either comma or period separator when the number is a float and a leading minus.

Parameters

char *	*str*	The string you want to check if is a number.

Returns
0 if *str* is numeric, 1 if it is not, -1 if an empty string or NULL value is passed.

See also
N/A

CslPrintf

Signature

```
char *      cslPrintf(char *fmt);
```

Description
The cslPrintf function expands the string in *fmt* which can have printf() value formatters like %s for string %d for integer etc. The function returns a string containing the expanded string, which is allocated. The caller is responsible for freeing the allocated memory.

Parameters

char *	fmt	String with variables in printf() format.
	…	Variable argument list

Returns
String from *fmt* containing the expanded variables defined on success, NULL if an error occurs.

See also
printf, free

Examples - Programming in the SLM Environment using C

Your first step in the development process is to decide which SDK you should acquire. The SDK is packed and shipped as a *probe package*, therefore available on the CA Nimsoft download page. Install the package on the system where you intend to develop the software. The package contains a code-generator (code wizard); please note that this is installed on Windows systems only.

The SDK contains the necessary library, and header files (depending on the language).

Header Files and Libraries

As with all Nimsoft probes written in C, you need to add the following two lines at the top of your code:

```
#include <nimbus.h>

#include <nim.h>
```

The linker needs references to the *nim.lib* (on windows) and *libnim.a* (UNIX).

The following are internal to the library and should not be called even though they are in the header file:

```
Char *cslStrTok(char *str, char *tokens); Char *cslStrTokDup(char *str, char
*tokens);
```

Sending a QoS Definition

The probe should always initialize itself by assuming that the *Data Engine* has no knowledge of the QoS data it is about to receive; therefore we need to send a QOS_DEFINITION message. Providing the nimQoSDefinition function with a correct and valid parameter list generates this message.

```
nimQoSDefinition   ("QOS_NET_CONNECT",              /* QOS Name*/
                   "QOS_NETWORK",                   /* QOS Group */
                   "Network Connectivity Response", /* QOS Description*/
                   "Milliseconds","ms",0,0);        /* Unit info*/
```

Sending Quality of Service Data

The probe should only initialize itself during startup by sending QOS_DEFINITION, however, it must report the collected data every time it runs. This instructs the *Data Engine* to insert the collected sample value into the database. The following code packs the QoS message into a function:

```
void publishQoS (char *target, long samplevalue)
{
        NIMQOS *qos      = NULL;
        Char *source     = NULL;       /*QoS origin (dhost)*/
        int interval     = 300;        /*Check interval (in seconds)*/

        if (!(qos = nimQoSCreate("QOS_NET_CONNECT",source,interval)))
        {
                nimLog(0,"(publishQoS) failed to create NIMQOS");
                return;
        }
        if (samplevalue < 0)
        {
                nimQoSSendNull (qos,target);
        }
        else
        {
                nimQoSSendValue(qos,target,samplevalue);
        }
        nimQoSFree(qos);
}
```

The following code illustrates how you may wrap your data collection code with stopwatch functionality--useful when doing response time measurements.

```
void checkMissionData ()
{
            NIMQOS      *qos           = NULL;
            char        *source        = NULL;      /*QoS origin (dhost) */
            char        *target        = "bandwidth_utilization";
            int         interval + 300;             /*Check interval (seconds)*/

            If (!(qos = nimQoSCreate("QOS_NET_CONNECT",source,interval)))
            {
                  nimLog(0,"(publishQoS) failed to create NIMQOS")
                  return;
            }
            nimQoSStart(qos);
                                                    /*Do your work…*/
            nimQoSStop(qos);
            nimQoSSendTimer (qos,target);           /*The timer is sent as QoS"*/
            nimQoSFree(qos);
}
```