CA Nimsoft Monitor

Perl SDK Reference Guide

v5.06 series

CA Nimsoft Monitor Copyright Notice

Contact CA

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: Perl SDK Overview

About Perl

Perl brings to CA Nimsoft Monitor a simple, yet powerful network programming environment. With it, users build everything from scripts generating simple alarms to powerful client/server solutions.

The Perl SDK Library

The Perl SDK modules wrap the Nimsoft Message Bus API functions, easing the development of probes that are written in Perl.

API.pm is the foundation Perl object, providing your interface into the Nimsoft Controller. Three other Perl objects are abstractions for making particular tasks more convenient:

- **CFG.pm** - for manipulating probe configuration files
- **PDS.pm** - for working with machine-independent data manipulation routines (PDS means **Portable Data Stream**)
- **Session.pm** - for using the functions available once a Nimsoft session is established.

Conventions Used

The following prefixes are used:

Prefix	Description
sz	string
h	handle
f	float/double
i	integer
b	boolean
cfg	configuration file
iRet	return code (see page 9) (integer); see table below for details.
ci	Configuration Item (see page 11)

Error Return Codes

NIME_OK	0

NIME_ERROR	1
NIME_COMERR	2
NIME_INVAL	3
NIME_NOENT	4
NIME_ISENT	5
NIME_ACCESS	6
NIME_AGAIN	7
NIME_NOMEM	8
NIME_NOSPC	9
NIME_EPIPE	10
NIME_NOCMD	11
NIME_LOGIN	12
NIME_SIDEXP	13
NIME_ILLMAC	14
NIME_ILLSID	15
NIME_SIDSESS	16
NIME_EXPIRED	17
NIME_NOLIC	18
NIME_INVLIC	19
NIME_ILLLIC	20
NIME_INVOP	21
NIME_USER	100

Configuration Item (CI)

A configuration item (CI)is an entity associated with a device. For example, a disk on a computer or an application running on a system

A CI must have a path (also known as "type"), which is a dotted-decimal path such as "1.1". The path / type is taken from a list available from CiManager.

A CI also has an optional name, which is a free-form string and can be something like "C:" (for a disk name) or "eth0" (for a network device name.) This device name is used to determine the uniqueness of a given CI. A CI unique identifier consists of the device being monitored, the path, and the name. When a remote system is involved, the device unique identifier incorporates the remote system IP address and the local (robot) system device identifier.

Perl SDK Prerequisites

The Perl SDK requires Nimsoft Server version 6.0 or later.

Additional SDK References

In addition to this guide, Nimsoft Perl modules are documented using standard Perl pod (Plain Old Documentation), typically accessed by issuing **perldoc Nimbus::API.**

Chapter 2: Perl SDK Reference

This section contains:

Perl API Library

nimInit

Signature

```
int  nimInit(iFlag);
```

Description

Initializes the Nimsoft Bus. This method is called upon loading; it is not necessary to call explicitly under normal circumstances.

Returns

iRet - The return code. Returns NIME_OK on success, or a NIME_{xxx} error code on failure, where {xxx} is the specific error code (see page 9).

nimAlarm

Signature

```
my($iRet, $szId) = nimAlarm($iLevel, $szMsg[, $szSub[, $szSup[, $szSrc]]]);
```

Description

Post an alarm message on the Nimsoft bus. The Level constants are:

NIML_CLEAR	0
NIML_INFO	1
NIML_WARNING	2
NIML_MINOR	3
NIML_MAJOR	4
NIML_CRITICAL	5

Parameters

Integer	*iLevel*	The alarm level (see Level constants above in Description)
String	*szMsg*	The alarm message
String	*szSub*	The subsystem identifier, e.g. 1.2.3 (default: 1.1)
String	*szSup*	The suppression definition (default: none)
String	*szSrc*	The alarm source (default: localhost).

Returns

- iRet - The return code. Returns NIME_OK on success, or a NIME_{xxx} error code on failure, where {xxx} is the specific error code (see page 9).
- szId - The Nimsoft Message ID for this alarm. The Nimsoft Message ID is used as the alarm identifier in the NAS.

nimGetIpToName

Signature

```
my($iRet, $szName) =  nimGetIpToName($szIp, $iPort);
```

Description

Map a host IP and port to the Nimsoft probe address.

Parameters

String	*szIp*	The hostname or host IP address of the target system
Integer	*iPort*	The port number of the targeted service

Returns

- iRet - The return code. Returns NIME_OK on success, or a NIME_{xxx} error code on failure, where {xxx} is the specific error code (see page 9).
- szName - The Nimsoft address (for example, the address for a probe or hub).

nimGetNameToIp

Signature

```
my($iRet, $szIp,$iPort) = nimGetNameToIp($szName);
```

Description

Maps a Nimsoft probe address to its host IP address and port.

Parameters

String	*szName*	The name (of the probe, or hub, etc.) in nimaddress form, meaning the full **/domain/hub/robot/probe** form can be specified, or the shortened version of **probe** if the query is meant to go to the controller running on the same robot as where the call is made.

Returns

- iRet - The return code. Returns NIME_OK on success, or a NIME_{xxx} error code on failure, where {xxx} is the specific error code (see page 9).
- szIp - The IP address of the targeted service.
- iPort - The port number of the targeted service.

nimGetVarStr

Signature

```
int nimGetVarStr($what, $data, $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.
String	*data*	Buffer to return value in.
int	*dlen*	Buffer Size.

Returns

- iRet - The return code. Returns NIME_OK on success, or a NIME_{xxx} error code on failure, where {xxx} is the specific error code (see page 9).
- The requested API value.

  Important! The NIMV_ROBOTNAME API value now returns the robotname, not the short host name.

nimLog

Signature

```
nimLog($iLevel, $szString);
```

Description

Log a message. nimLogSet sets the logfile.

Parameters

Int	*iLevel*	The loglevel (levels <= iLevel are logged).
String	*szString*	The message to write to log.

Returns

Void

nimLogClose

Signature

```
nimLogClose();
```

Description

Close the current log.

Parameters

None

Returns

Void

nimLogSet

Signature

```
nimLogSet($szFile, $szPrefix, $iLevel, $iFlags);
```

Description

Initialize the logging system.

Parameters

String	*szFile*	The logfile name (or 'stdout').
String	*szPrefix*	The message prefix (for a multiplexed log).
Integer	*iLevel*	The current loglevel (levels <= iLevel are logged).
Integer	*iFlags*	Currently unsupported.

Returns

iRet - The return code. Returns NIME_OK on success, or a NIME_{xxx} error code on failure, where {xxx} is the specific error code (see page 9).

nimLogSetLevel

Signature

```
nimLogSetLevel($iLevel);
```

Description

Set new loglevel.

Parameters

Integer	*iLevel*	The new loglevel.

Returns

Integer value of the original loglevel (typically of low interest).

nimNamedRequest

Signature

```
my($iRet, $retdata) = nimNamedRequest($szAddr, $szCmd, $udata, $iSec);
```

Description

Send a named request over the Nimsoft Bus to a server. This function works through Hub Tunnels, because the routing of messages is up to the Hub.

Parameters

String	*szAddr*	The full Nimsoft address.
String	*szCmd*	The service command.
PDS	*udata*	The PDS record with user data.
Integer	*iSec*	Time in seconds to wait for reply.

Returns

- iRet - The return code. Returns NIME_OK on success, or a NIME_{xxx} error code on failure, where {xxx} is the specific error code (see page 9).
- retdata - The return PDS record.

See Also

NimRequest (see page 23)

nimPostMessage

Signature

```
my($iRet, $szId) = nimPostMessage($szSubject, $iLevel, $szSup, $udata);
```

Description

Post a user-defined message on the Nimsoft bus.

Parameters

String	*szSubject*	The subject identifier for the posted message.
Integer	*iLevel*	The post priority.
String	*szSup*	The suppression definition (default: none).
PDS	*udata*	A PDS record with user data.

Returns

- iRet - The return code. Returns NIME_OK on success, or a NIME_{xxx} error code on failure, where {xxx} is the specific error code (see page 9).
- szId - The Nimsoft Message ID for this message.

nimQoSDefinition

Signature

```
nimQoSDefinition($szQoSName, $szQoSGroup, $szDescription, $szUnit, $szAbbr,
$iHasMax, $iIsBool);
```

Description

Deprecated--use nimQoSSendDefinition (see page 21).

Sends a QoS definition to the SLM subsystem. Creates a new QoS data type if the definition has not been sent previously. Must be sent (once) before sending the QoS messages for this data type.

Parameters

String	*szQoSName*	The name of the QoS--in the form *QOS_xxx.*
String	*szQoSGroup*	The name of the logical group that the new QoS will belong to--in the form *QOS_xxx.*
String	*szDescription*	A description of the QoS object.
String	*szUnit*	The unit type of the QoS data (milliseconds, kilobytes).
String	*szAbbr*	The unit short form or abbreviation.
Integer	*iHasMax*	If the QoS object has a maximum value, e.g. disk size; memory usage.

Integer	*iisBool*	Is data type Boolean (reports only 1=true or 0=false). Examples: host is available; printer is unavailable.

Returns

iRet - The return code. Returns NIME_OK on success, or a NIME_{xxx} error code on failure, where {xxx} is the specific error code (see page 9).

nimQoSCreate

Signature

```
nimQoSCreate($szQoSName, $szTarget, $samplerate, $samplemax);
```

Description

Creates and initializes a QoS object and returns a handle to this object.

Parameters

String	*szQoSName*	The name of the QoS.
String	*szTarget*	The QoS target.
Integer	*samplerate*	The rate of the sample interval.
Integer	*samplemax*	The maximum value (if any). For example: disksize.

Returns

hQoS - handle to the QoS object.

nimQoSFree

Signature

```
nimQoSFree($handle);
```

Description

Release the resources that are allocated in the handle.

Parameters

Handle	*Handle*	The QoS handle.

Returns

N/A

nimQoSMessage

Signature

```
my($iRet, $szId) = nimPostMessage($szSubject, $iLevel, $szSup, $udata);
```

Description

Deprecated--use nimQoSPostMessage (see page 20).

Posts a user-defined message on the Nimsoft bus.

Parameters

String	*szSubject*	The subject identifier for the posted message.
Integer	*iLevel*	The post priority.
String	*szSup*	The suppression definition (default: none).
PDS	*udata*	A PDS record with user data.

Returns

- iRet - The return code. Returns NIME_OK on success, or a NIME_{xxx} error code on failure, where {xxx} is the specific error code (see page 9).
- szId - The Nimsoft Message ID for this message.

nimQoSPostMessage

Signature

```
int nimQoSPostMessage($handle, $sztarget, $fsamplevalue, $fsample_stdev)
```

Description

Replaces the deprecated nimQoSMessage (see page 20). Sends a raw QoS message.

Parameters

Handle	*handle*	QoS handle
String	*sztarget*	QoS target
Float	*fsamplevalue*	Value of the QoS measurement
Float	*fsample_stdev*	The QoS measurement's standard deviation.

Returns

iRet - The return code. Returns NIME_OK on success, or a NIME_{xxx} error code on failure, where {xxx} is the specific error code (see page 9).

nimQoSSendDefinition

Signature

```
my($iRet) = nimQoSSendDefinition($szName, $szGroup, $szDesc, $szUnit, $szAbbr
[,$iFlags]);
```

Description

New version of the deprecated nimQoSDefinition (see page 18) function; sends a QoS definition to the Nimsoft bus. This definition must be in place before sending QoS messages for the given named QoS. Should only be sent on startup to avoid excessive work for the data_engine probe.

Parameters

String	*szName*	The name of the QoS.
String	*szGroup*	The name of the logical group that the new QoS will belong to.
String	*szDesc*	A description of the QoS object.
String	*szUnit*	The unit type of the QoS data (milliseconds, kilobytes).
String	*szAbbr*	The unit short form or abbreviation (MB).
	iFlags	Optional flags: NIMQOS_DEF_NONE NIMQOS_DEF_BOOLEAN NIMQOS_DEF_HASMAX NIMQOS_DEF_ASYNCH NIMQOS_DEF_REDEFINE

Returns

iRet - The return code. Returns NIME_OK on success, or a NIME_{xxx} error code on failure, where {xxx} is the specific error code (see page 9).

nimQoSSendNull

Signature

```
int nimQoSSendNull($Handle, $sztarget);
```

Description

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

Parameters

Handle	*Handle*	QoS handle
String	*sztarget*	QoS target

Returns

iRet - The return code. Returns NIME_OK on success, or a NIME_{xxx} error code on failure, where {xxx} is the specific error code (see page 9).

nimQoSSendValue

Signature

```
nimQoSSendValue($hQoS, $szSource, $iValue);
```

Description

Send a QoS value.

Parameters

String	*hQoS*	The name of the QoS.
String	*szSource*	The QoS target.
Integer	*iValue*	The rate of the sample interval.

Returns

iRet - The return code. Returns NIME_OK on success, or a NIME_{xxx} error code on failure, where {xxx} is the specific error code (see page 9).

nimQoSStart

Signature

```
nimQoSStart($hQoS);
```

Description

Sets the start time in the given QoS object.

Parameters

QoShandle	*hQoS*	The QoS handle returned by nimQoSCreate.

Returns

iRet - The return code. Returns NIME_OK on success, or a NIME_{xxx} error code on failure, where {xxx} is the specific error code (see page 9).

nimQoSStop

Signature

```
nimQoSStop($hQoS);
```

Description

Sets the stop timer in the given QoS object and calculates the time used since nimQoSStart was run.

Parameters

QoShandle	*hQoS*	The QoS handle returned by nimQoSCreate.

Returns

Void

nimQoSSendTimer

Signature

```
nimQoSSendTimer($hQoS, $szSource);
```

Description

Sends a QoS message with the time (in milliseconds). The time is the interval between when nimQoSStart was called and nimQoSStop was called. If nimQoSStop has not been called the time between when nimQoSStart was called and the current time is used.

Parameters

QoShandle	*hQoS*	The QoS handle returned by nimQoSCreate.
String	*szSource*	The QoS Source.

Returns

iRet - The return code. Returns NIME_OK on success, or a NIME_{xxx} error code on failure, where {xxx} is the specific error code (see page 9).

nimRequest

Signature

```
my($iRet, $retdata) = nimRequest($szAddr, $iPort, $szCmd, $udata, $iSec);
```

Description

Send a request over the Nimsoft Bus to a server. This function does not traverse hub tunnels, because it attempts a direct connection to the given IP/port. A firewalled environment causes difficulties.

Parameters

String	*szAddr*	The hostname or host IP address.

Integer	*iPort*	The service port.
String	*szCmd*	The service command.
PDS	*udata*	The PDS record with user data.
Integer	*iSec*	Time in seconds to wait for reply.

Returns

- iRet - The return code. Returns NIME_OK on success, or a NIME_{xxx} error code on failure, where {xxx} is the specific error code (see page 9).
- retdata - The return PDS record.

See Also

- nimNamedRequest (see page 17)
- nimGetNameToIp (see page 15)

nimSuppToStr

Signature

```
my($szSup) = nimSuppToStr($bHold, $iNumber, $iSeconds, $szSuppKey);
```

Description

Creates a suppression definition string.

Parameters

Boolean	*bHold*	Send and then suppress.
Integer	*iNumber*	Suppress n messages.
Integer	*iSeconds*	Within n seconds.
String	*szSuppKey*	Suppression key.

Returns

szSup - The suppression definition (default: none).

Examples

```
$szSup = nimSuppToStr(0,0,0,"FileSystem|$name");
$szSup = nimSuppToStr(1,0,60,"");
```

cfgClose

Signature

```
my($iRet) = cfgClose($cfg);
```

Description

Close the config file.

Parameters

CFGhandle	*cfg*	A pointer to a structure containing the configuration data.

Returns

iRet - The return code. Returns NIME_OK on success, or a NIME_{xxx} error code on failure, where {xxx} is the specific error code (see page 9).

cfgKeyDelete

Signature

```
my($iRet) = cfgKeyDelete($cfg, $szSection, $szKey);
```

Description

Delete a key in a section of the config file.

Parameters

CFGhandle	*cfg*	A pointer to a structure containing the configuration data.
String	*szSection*	The name of the section to which the key belongs.
String	*szKey*	The name of the key to be deleted.

Returns

iRet - The return code. Returns NIME_OK on success, or a NIME_{xxx} error code on failure, where {xxx} is the specific error code (see page 9).

cfgKeyList

Signature

```
my($list) = cfgKeyList($cfg, $szSection);
```

Description

List an array of keys in the specified section of a configuration file.

Parameters

CFGhandle	*cfg*	A pointer to a structure containing the configuration data.

String	*szSection*	The name of the section to which the key belongs.

Returns

An array containing the keys in the specified section.

cfgKeyRead

Signature

```
my($szValue) = cfgKeyRead($cfg, $szSection, $szKey);
```

Description

Read a value from a configuration file.

Parameters

CFGhandle	*cfg*	A pointer to a structure containing the configuration data.
String	*szSection*	The name of the section to which the key belongs.
String	*szKey*	The name of the key.

Returns

szValue - The value that is assigned to the specified key.

cfgKeyRename

Signature

```
my($iRet) = cfgKeyRename($cfg, $szSection, $szOld, $szNew);
```

Description

Rename a key in a section of the configuration file.

Parameters

CFGhandle	*cfg*	A pointer to a structure containing the configuration data.
String	*szSection*	The name of the section to which the key belongs
String	*szOld*	The old name of the key.
String	*szNew*	The new name of the key.

Returns

iRet - The return code. Returns NIME_OK on success, or a NIME_{xxx} error code on failure, where {xxx} is the specific error code (see page 9).

cfgKeyWrite

Signature

```
my($iRet) = cfgKeyWrite($cfg, $szSection, $szKey, $szValue);
```

Description

Write the specified key/value pair to the configuration file in the given section.

Note: For nested sections, specify the full path to the section. For example, "/profiles/profile1"

Parameters

CFGhandle	*cfg*	A pointer to a structure containing the configuration data.
String	*szSection*	The name of the section to which the key belongs.
String	*szKey*	The name of the key.
String	*szValue*	The value of the key.

Returns

iRet - The return code. Returns NIME_OK on success, or a NIME_{xxx} error code on failure, where {xxx} is the specific error code (see page 9).

cfgListRead

Signature

```
my($fsList) = cfgListRead($cfg, $szSection);
```

Description

Reads the section that is specified and returns the list of values as an array.

Parameters

CFGhandle	*cfg*	A pointer to a structure containing the configuration data.
String	*szSection*	The name of the section to add the table values to.

Returns

An array containing the values that are found in the section.

cfgListWrite

Signature

```
my($iRet) = cfgListWrite($cfg, $szSection, $szKeyBody, $List);
```

Description

Replace a section with the contents of the array list. Each entry in the array is treated as a value. The key names are generated based on the KeyBody prefix (for example, "key" generates "key0", "key1"…).

Parameters

CFGhandle	*cfg*	A pointer to a structure containing the configuration data.
String	*szSection*	The name of the section to add the table values to.
String	*szKeyBody*	The basis (prefix) name to use when generating keys.
Array	*List*	The array containing values that are to be added.

Returns

iRet - The return code. Returns NIME_OK on success, or a NIME_{xxx} error code on failure, where {xxx} is the specific error code (see page 9).

cfgOpen

Signature

```
my($cfg) = cfgOpen($szFile, $bReadOnly);
```

Description

Opens and reads the config file, returning a CFG object with the config data.

Parameters

String	*szFile*	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
Boolean	*bReadOnly*	Opens the file in read mode if TRUE, in read-write mode if FALSE.

Returns

cfg - A handle to a CFG object containing the configuration data found in the configuration file.

cfgSectionCopy

Signature

```
my($iRet) = cfgSectionCopy($cfg, $szFrom, $szTo);
```

Description

Copies a section to a new name at the same level.

Parameters

CFGhandle	*cfg*	A pointer to a structure containing the configuration data.
String	*szFrom*	The name of the section to be copied.
String	*szTo*	The new name of the section to where the copied contents will be pasted.

Returns

iRet - The return code. Returns NIME_OK on success, or a NIME_{xxx} error code on failure, where {xxx} is the specific error code (see page 9).

cfgSectionDelete

Signature

```
my($iRet) = cfgSectionDelete($cfg, $szSection);
```

Description

Delete the specified section.

Parameters

CFGhandle	*cfg*	A pointer to a structure containing the configuration data.
String	*szSection*	The name of the section to be deleted.

Returns

iRet - The return code. Returns NIME_OK on success, or a NIME_{xxx} error code on failure, where {xxx} is the specific error code (see page 9).

cfgSectionList

Signature

```
my($list) = cfgSectionList($cfg, $section);
```

Description

Read the array of sections in a config file.

Parameters

CFGhandle	*cfg*	A pointer to a structure containing the configuration data.
String	*section*	The section whose underlying sections are to be listed.

Returns

list - an array containing the list of sections under the specified section.

cfgSectionRename

Signature

```
my($iRet) = cfgSectionRename($cfg, $szOld, $szNew);
```

Description

Rename the specified section.

Parameters

CFGhandle	*cfg*	A pointer to a structure containing the configuration data.
String	*szOld*	The old name of the section to be renamed.
String	*szNew*	The new name of the section to be renamed.

Returns

iRet - The return code. Returns NIME_OK on success, or a NIME_{xxx} error code on failure, where {xxx} is the specific error code (see page 9).

cfgSynch

Signature

```
my($iRet) = cfgSync($cfg);
```

Description

Write the configuration file (buffer) to disk, but leave the file open.

Parameters

CFGhandle	*cfg*	A pointer to a structure containing the configuration data.

Returns

iRet - The return code. Returns NIME_OK on success, or a NIME_{xxx} error code on failure, where {xxx} is the specific error code (see page 9).

ciAlarm

Signature

```
my($iRet,$szId) = ciAlarm($handle, $szMetric, $iLevel, $szMsg[,
$pdsAlarmValues[, $szSubsystem
[, $szSuppression[, $szSource[, $szCustom11[, $szCustom2[, $szCustom3[,
$szCustom4[, $szCustom5]]]]]]]]);
```

Description

Post an alarm message on a change in CI status or QoS.

Note: If you need to send more than one alarm within a short period of time, use the ciSessionAlarm (see page 35) function instead.

Parameter

CIHANDLE	*handle*	The handle reference returned from other functions such as ciOpenLocalDevice().
String	*szMetric*	The name of the metric.
Integer	*iLevel*	The alarm level.
String	*szMsg*	The alarm message.

Optional Parameters

PDS	*pdsAlarmValues*	The PDS object.
String	*szSubsystem*	The subsystem identifier. For example, 1.2.3.
String	*szSuppression*	The suppression definition.
String	*szSource*	The alarm source.
String	*szCustom1*	The "Custom 1" alarm property.
String	*szCustom2*	The "Custom 2" alarm property.
String	*szCustom3*	The "Custom 3" alarm property.
String	*szCustom4*	The "Custom 4" alarm property.
String	*szCustom5*	The "Custom 5" alarm property.

Returns

- iRet - The return code. Returns NIME_OK on success, or a NIME_{xxx} error code on failure, where {xxx} is the specific error code (see page 9).
- szId - The Nimsoft Message ID for this alarm. The Nimsoft Message ID is used as the alarm identifier in the NAS.

ciAttributeNumber

Signature

```
my($iRet) = ciAttributeNumber($handle, $szKey, $iValue);
```

Description

Sets attributes as key/value pairs in a CI handle.

Parameters

CIhandle	*handle*	The handle reference returned from other functions, such as ciOpenLocalDevice().
Character	*szKey*	The name of the key.
double	*iValue*	The value of the key.

Returns

iRet - The return code. Returns NIME_OK on success, or a NIME_{xxx} error code on failure, where {xxx} is the specific error code (see page 9).

ciAttributeString

Signature

```
my($iRet) = ciAttributeString($handle, $szKey, $szValue);
```

Description

Sets attributes as key/value pairs in a CI handle.

Parameters

CIhandle	*handle*	The handle reference returned from other functions, such as ciOpenLocalDevice().
Character	*szKey*	The name of the key.
double	*szValue*	The key value.

Returns

iRet - The return code. Returns NIME_OK on success, or a NIME_{xxx} error code on failure, where {xxx} is the specific error code (see page 9).

ciBindQoS

Signature

```
my($iRet) = ciBindQoS($handle, $hQoS, $szMetric);
```

Description

Creates a binding to a CIHANDLE in the QOSHANDLE for a given metric.

Parameters

CIhandle	*handle*	The handle reference previously returned from other functions such as ciOpenLocalDevice().
NIMQOS	*hQoS*	The QoS handle.
Character	*szmetric*	The name of the metric.

Returns

iRet - The return code. Returns NIME_OK on success, or a NIME_{xxx} error code on failure, where {xxx} is the specific error code (see page 9).

ciClose

Signature

```
my($iRet) = ciClose($handle);
```

Description

Releases the CI handle.

Parameters

CIHANDLE	*handle*	The handle reference previous returned from other functions, such as ciOpenLocalDevice().

Returns

iRet - The return code. Returns NIME_OK on success, or a NIME_{xxx} error code on failure, where {xxx} is the specific error code (see page 9).

ciGetCachePath

Signature

```
my($Ret, $szPath) = ciGetCachePath();
```

Description

Returns the path to the CI cache in the file system.

Returns

- iRet - The return code. Returns NIME_OK on success, or a NIME_{xxx} error code on failure, where {xxx} is the specific error code (see page 9).
- szPath - A path to the CI cache in the file system.

ciOpenLocalDevice

Signature

```
my($hCI) = ciOpenLocalDevice($szType, $szName);
```

Description

Gets the CI handle object for a local device.

Parameters

String	*szType*	The CI Type.
String	*szName*	The CI Name.

Returns

hCI - a CI handle.

ciOpenRemoteDevice

Signature

```
my($hCI) = ciOpenRemoteDevice($szType, $szName, $szHost);
```

Description

Gets the CI handle for a remote device.

Parameters

String	*szType*	ciType.
String	*szName*	ciName.
String	*szHost*	Host for which to create a lookup.

Returns

hCI - a CI handle.

ciRelationship

Signature

```
my($iRet) = ciRelationship($hCIParent, $hCIChild);
```

Description

Sets a parent/child relationship between two CI handles.

Parameters

CIhandle	*hCIParent*	Parent CI handle object.
CIhandle	*hCIChild*	Child CI handle object.

Returns

iRet - The return code. Returns NIME_OK on success, or a NIME_{xxx} error code on failure, where {xxx} is the specific error code (see page 9).

ciSessionAlarm

Signature

```
my($iRet,$szId) = ciSessionAlarm($nims, $hCI, $szMetric, $iLevel, $szMsg[,
$pdsAlarmValues
[, $szSubsystem[, $szSuppression[, $szSource[, $szCustom11[, $szCustom2[,
$szCustom3
[, $szCustom4[, $szCustom5]]]]]]]]);
```

Description

Post an alarm message on a change in CI status or QoS. This function behaves exactly like the ciAlarm (see page 31) function, except that a session is required before sending the alarm. ciSessionAlarm is useful to send more than one alarm within a short time.

Parameter

NIMSESS	*nimsess*	The session object to log information for.
CIHANDLE	*handle*	The handle reference returned from other functions such as ciOpenLocalDevice().
String	*szMetric*	The name of the metric.
Integer	*iLevel*	The alarm level.
String	*szMsg*	The alarm message.

Optional Parameters

PDS	*pdsAlarmValues*	The PDS object.
String	*szSubsystem*	The subsystem identifier. For example, 1.2.3.
String	*szSuppression*	The suppression definition.
String	*szSource*	The alarm source.

String	*szCustom1*	The "Custom 1" alarm property.
String	*szCustom2*	The "Custom 2" alarm property.
String	*szCustom3*	The "Custom 3" alarm property.
String	*szCustom4*	The "Custom 4" alarm property.
String	*szCustom5*	The "Custom 5" alarm property.

Returns

- iRet - The return code. Returns NIME_OK on success, or a NIME_{xxx} error code on failure, where {xxx} is the specific error code (see page 9).
- szId - The Nimsoft Message ID for this alarm.

ciUnBindQoS

Signature

my($iRet) = ciUnBindQoS($hQoS);

Description

ciUnBindQoS removes any binding in the NIMQOS handle.

Parameters

NIMQOS	*hQoS*	The QoS object to be un-bound.

Returns

iRet - The return code. Returns NIME_OK on success, or a NIME_{xxx} error code on failure, where {xxx} is the specific error code (see page 9).

cslMatchRegExp

Signature

```
my($iRet) = cslMatchRegExp($szTargetString, $szMatchExpr);
```

Description

Check string with pattern matching or regular expression string.

Parameters

String	*szTargetString*	Input string to match against.
String	*szMatchExpr*	Regular expression to be applied against szTargetString.

Returns

iRet - 1 on success, 0 on a failure to match the MatchExpr in TargetString.

Note: Perl has regular expression support that is built into the language, so this function is here for compatibility with the C libraries.

Example

```
if (cslMatchRegExp("help me now","*me*")) {
print "Found match...";
    }
```

gettimeofday

Signature

```
my($sec, $usec) = gettimeofday();
```

Description

Returns the time

Returns

sec and usec - returns the time in seconds and in micro-seconds if in a scalar context, or returns an array with 2 values (seconds, microseconds).

Example

```
my @start = gettimeofday();
# do stuff...
my @end = gettimeofday();
```

tv_interval

Signature

```
my $diff = tv_interval(@start, @end);
```

Description

Calculates the difference between the start and end times and returns this difference as a number of microseconds used. Used for high-resolution timers, giving greater granularity than normal timers.

Parameters

start	Array from gettimeofday()
end	Array from gettimeofday()

Returns

The differential between the specified start and end times in microseconds.

The Configuration Library

The CFG object is a class wrapper around the functions that are targeted against configuration files. The relevant functions are listed in the CFG.pm module.

Nimbus::API::cfg*

When a new CFG object is constructed the constructor takes one required argument (the configuration filename), and one optional (a 'private' hash). It is normal to maintain the hash within the CFG object, but in some cases it can be useful to add the configuration data to a private hash.

Synopsis

use Nimbus::CFG

my $cfg = Nimbus::CFG->new(["my.cfg" [,$hptr]]); (see page 39)

$cfg->debug($boolean); (see page 39)

$cfg->dump($cfg); (see page 40)

$cfg->getKeys($hptr); (see page 40)

$cfg->getSections($hptr); (see page 40)

$cfg->getValues($hptr); (see page 41)

$cfg->open("my.cfg" [,$hptr]); (see page 41)

$cfg->setConverter(\&src [,\&dst]); (see page 42)

Example

When two or more configuration files are referenced by one hash:

Call this configuration 'test.cfg':

```
<setup>
logfile = stdout
loglevel = 2
        <names>
                name_0 = luke
                name_1 = leia
                name_2 = r2d2
        </names>
</setup>
```

The following code accesses the values from the setup section:

```
use Nimbus::CFG;
my $cfg = Nimbus::CFG->new("test.cfg");
print "The logfile : $cfg->{setup}->{logfile}\n";
print "The loglevel: $cfg->{setup}->{loglevel}\n";
```

new

Signature

```
new("my.cfg" [,$hptr]);
```

Description

The **new** method takes a filename/path as a required parameter, and a hashptr as an optional parameter.

Parameters

String	*file*	The name of the file to create.
Hashpointer	*hptr*	Pointer to a hash to populate.

Returns

A reference to the Nimbus::CFG object.

debug

Signature

```
debug($boolean);
```

Description

Sets the debug flag. If debug has been set to True (1), then extra information is printed on stdout.

Parameters

Integer	*boolean*	0 or 1

Returns

iRet - The return code. Returns NIME_OK on success, or a NIME_{xxx} error code on failure, where {xxx} is the specific error code (see page 9).

dump

Signature

```
dump($cfg);
```

Description

Prints the contents of the config to stdout.

Parameters

Hashpointer	*cfg*	Optional pointer to the hash with the configuration data.

Returns

iRet - The return code. Returns NIME_OK on success, or a NIME_{xxx} error code on failure, where {xxx} is the specific error code (see page 9).

getKeys

Signature

```
my(@arrKeys) = getKeys($hashPtr);
```

Description

Returns an array of keys in the hash.

Parameters

Hashpointer	*hashPtr*	Optional hashpointer.

Returns

arrKeys - An array of keys.

getSections

Signature

```
my(@arrkeys) = getSections($hptr);
```

Description

Returns an array of section names in the hash.

Parameters

Hashpointer	*hptr*	Optional hashpointer.

Returns

arrKeys - An array of section names in the configuration file.

getValues

Signature

```
my(@arrKeys) = getValues($hptr);
```

Description

The getValues method takes a hashptr as its input parameter, and returns an array of the values that are taken from each key/value pair in the section.

Parameters

Hashpointer	*hptr*	Optional parameter, points to the hash to read from

Returns

arrKeys - An array of values in the hash.

Example

Use the configuration file 'test.cfg' from the DESCRIPTION, and the following code segment to extract and access the data:

```
use Nimbus::CFG;
use strict;

my $cfg   = Nimbus::CFG->new("test.cfg");
my @names = $cfg->getValues($cfg->{setup}->{names});
```

@names now holds 3 names.

open

Signature

```
open("my.cfg"[, $hptr]);
```

Description

The open method takes a filename (or path) as a required parameter, and a hashptr as an optional parameter.

- This method is used when postponed parsing is needed, for instance when setting the name converter before parsing the file.
- In effect, **open** delivers the same result as **new** does.
- This method calls the underlying function cfgopen (see page 28).

Parameters

String	*file*	The name of the file to open.

Hashpointer	*hptr*	Pointer to a hash to populate.

Returns

A hash containing the contents of the configuration file.

setConverter

Signature

```
setConverter(\&src [,\&dst]);
```

Description

The setConverter method takes a reference to a function as a parameter. This function is called whenever a new section is parsed. The default converter substitutes every hash(#) in a section name with a slash(/). This can be useful when using the slash (/) character as part of the section/key name, such as a filename or equivalent.

Parameters

Function	*&src*	Function which translates keys on read.
Function	*&dst*	Function which translates keys on write.

Returns

Void

Example

Consider the following code and configuration file:

my.cfg:

```
<filesystems>
  <#dev#dsk#c0t3d0s4>
    name = /usr
        high = 99
        low  = 70
  </#dev#dsk#c0t3d0s4>
</filesystems>
```

```
my_wo_converter.pl:
#####################################################
# Script using the standard/builtin converter

use Nimbus::CFG;
my $cfg = Nimbus::CFG->new("my.cfg");

my $fs1 = $cfg->{filesystems}->{'/dev/dsk/c0t3d0s4'};

print "filesystem1: $fs1->{name}, high:$fs1->{high} \n";

==> will print 'filesystem1: /usr, high:99'
my_w_private_conv.pl:
#####################################################
# Script using a private converter

use Nimbus::CFG;

sub myconv {
  my $s = shift;
  $$s =~ s/\#/\>/g;   # convert from hash(#) to GT(>)
}

my $cfg = Nimbus::CFG->new();

$cfg->setConverter(\&myconv);
$cfg->open("my.cfg");
my $fs1 = $cfg->{filesystems}->{'>dev>dsk>c0t3d0s4'};

print "filesystem1: $fs1->{name}, high:$fs1->{high} \n";

==> will print 'filesystem1: /usr, high:99'.
```

PDS-Object Interface Wrapping the PDS

The PDS object is a class wrapper around the Nimbus::API PDS functions. Reference the PDS.pm module for the functions included.

Synopsis

use Nimbus::PDS

my $pds = Nimbus::PDS->new([$pdsData]); (see page 44)

$hptr = $pds->asHash(); (see page 45)

$pds->data(); (see page 45)

$pds->dump(); (see page 46)

$pds->float ($name, $value); (see page 46)

$value = $pds->get ($name [,$type]); (see page 46)

$value = $pds->getTable ($name [,$type]); (see page 47)

$pds->number($name, $value); (see page 47)

$pds->put ($name,$value [,$type]); (see page 48)

$pds->putTable ($name,$value [,$type]); (see page 48)

$pds->remove($name); (see page 48)

$pds->reset(); (see page 49)

$pds->rewind(); (see page 49)

$pds->string($name, $value); (see page 49)

new

Signature

```
new([$pdsData]);
```

Description

The new method takes pdsData as an optional parameter.

Parameters

PDS	*pdsData*	Scalar reference object returned from another API call.

Returns

A reference to a Nimbus::PDS object.

ashash

Signature

```
put ($name, $value[, $type]);
```

Description

The asHash method produces an associative array (hash) by traversing the PDS. If the PDS contains other PDSes, then the hierarchy is preserved by nesting.

Parameters

String	*name*	Key
String	*value*	Variable
PDS type	*type*	One of PDS_PCH, PDS_INT, PDS_F, PDS_PDS.

Returns

A hash containing the contents of the PDS.

Example

```
use Nimbus::PDS;
my $pds = Nimbus::PDS->new();
$pds->string("name", "Jane S");
$pds->number("age", 60);

my $h = $pds->asHash();
print "name: $h->{name}, age: $h->{age}\n";
```

data

Signature

```
data()
```

Description

The PDS package is a wrapper over the raw PDS functions in the library. One of the elements in the structure is a PDS that can be sent by nimRequest and similar functions. The data() function returns the reference to this PDS.

Parameters

None

Returns

The PDS element that the Nimbus::PDS structure wraps.

dump

Signature

```
dump()
```

Description

Prints the contents of the PDS to the logfile. NimLogSet must have been executed before this call is made.

Parameters

None

Returns

Prints the contents of the PDS to the logfile.

float

Signature

```
float ($name, $value);
```

Description

Adds the key/value pair as an element in the PDS. Wrapper for put() with the correct type set.

Parameters

String	*name*	Key
Float	*value*	Value

Returns

1 on success, 0 on failure.

get

Signature

```
get ($name[, $type]);
```

Description

Generic method for reading a value from a PDS for a given key. User must specify the *type* of value to read.

Parameters

String	*name*	Key
PDS type	*type*	One of PDS_PCH, PDS_INT, PDS_F, PDS_PDS.

Returns

The value of the specified key.

getTable

Signature

```
getTable ($name[, $type]);
```

Description

Returns the contents of a table in the PDS.

Parameters

String	*name*	Key
PDS type	*type*	One of PDS_PCH, PDS_INT, PDS_F, PDS_PDS.

Returns

Either an array, or a new PDS structure, depending on the type. If the operation fails nothing is returned.

number

Signature

```
number($name, $value);
```

Description

Adds the key/value pair as an element in the PDS. Wrapper for put() with the correct type set.

Parameters

String	*name*	Key
Integer	*value*	Value

Returns

1 on success, 0 on failure.

put

Signature

```
put($name, $value[, $type]);
```

Description

Adds the key/value pair as an element in the PDS.

Parameters

String	*name*	Key
Variable	*value*	Value, where type is specified in *type.*
PDS type	*type*	One of PDS_PCH, PDS_INT, PDS_F, PDS_PDS.

Returns

1 on success, 0 on failure.

putTable

Signature

```
my($bRet) = putTable($name, $value[, $type]);
```

Description

Adds the key/value pair as a table element in the PDS

Parameters

String	*name*	Key
Variable	*value*	Value, where type is specified in *type.*
PDS type	*type*	One of PDS_PCH, PDS_INT, PDS_F, PDS_PDS.

Returns

bRet = 1 on success, 0 on failure.

remove

Signature

```
remove($name);
```

Description

This function removes the element that is named 'key' from the PDS stream.

Parameters

String	*name*	Key to remove from the PDS.

Returns

PDS_ERR on failure, PDS_ERR_NONE on success.

reset

Signature

```
reset()
```

Description

This function re-initializes the PDS object to an empty buffer with initial pointer settings. If you want to reset the get pointer, use the pdsRewind() function. This function deletes data. The buffer is not reallocated.

Parameters

None

Returns

0 on success, non-zero value on failure.

rewind

Signature

```
rewind()
```

Description

This function re-initializes the get pointer within the PDS object.

Note: You can put a block of data and perform multiple gets separated with a pdsRewind(pds) call.

Parameters

None

Returns

PDS_ERR on failure, PDS_ERR_NONE on success.

string

Signature

```
string($name, $value);
```

Description

Adds the key/value pair as an element in the PDS. Wrapper for put() with the correct type set.

Parameters

String	*name*	Key
String	*value*	Value

Returns

1 on success, 0 on failure.

Session-Object Interface to the Nimsoft Bus

The Session object is a class wrapper around the Nimbus::API module, and raises the abstraction layer from the low-level Nimsoft API. Reference the Session.pm module for the functions included.

You can create a server (TCP/IP), that accepts commands over the ports registered by the $nim->*server* method. The command is dispatched by the command dispatcher to the function with the same name as the command. A command without a matching function causes an abort situation.

Another feature of this class is the ability to connect to a Nimsoft hub. The functions $nim->*attach* and $nim->*subscribe* both connect to the hub and receive postings over the **hubpost** function.

SYNOPSIS

use Nimbus::Session

my $nim = Nimbus::Session->new($id[, $session]); (see page 51)

$nim->addCallback($command[, $format[, $security_level]]); (see page 51)

$nim->attach($queue [, hubip [, hubport]]); (see page 52)

$nim->detach([$id]); (see page 52)

$nim->dispatch($timeout_ms [, $breakOnEvent]); (see page 53)

$nim->server([$port[, $timeoutCB [, $restartCB]]]); (see page 53)

$nim->setInfo($version[, $company]); (see page 54)

$nim->setPostCallback($function_name); (see page 54)

$nim->setRetryInterval($intervalAsSeconds); (see page 54)

$nim->subscribe($subjects [, hubip [, hubport]]); (see page 55)

new

Signature

```
new($id[, [$session]);
```

Description

The **new** method takes a session ID as a required parameter, and session name as an optional parameter.

Parameters

Integer	*id*	session ID.
String	*session*	optional session name.

Returns

szSessionId - a reference to the Nimbus::Session object.

addCallback

Signature

```
addCallback($command[, $format[, $security_level]]);
```

Description

Adds a callback to the server session. This allows functions to be triggered from other probes or utilities.

Parameters

String	*command*	Name of the function to be called when callback is triggered.
String	*format*	Parameters to the function and their data types.
Integer	*security_level*	0=open, 1=read, 2=write, 3=admin.

Returns

iRet - The return code. Returns NIME_OK on success, or a NIME_{xxx} error code on failure, where {xxx} is the specific error code (see page 9).

attach

Signature

```
attach($queue[, hubip[, hubport]]);
```

Description

The attach method causes the session to attach to a queue defined on the hub. As opposed to the subscribe (see page 55) method, messages are queued when the process is detached (for example, not running, or running but detached).

Parameters

String	*queue*	Comma separated list of subjects to listen for on the Hub.
String	*hubip*	Optional IP address of the Hub.
Integer	*hubport*	Optional Port on which the Hub is listening (48002).

Returns

iRet - The return code. Returns NIME_OK on success, or a NIME_{xxx} error code on failure, where {xxx} is the specific error code (see page 9).

detach

Signature

```
detach([$id]);
```

Description

The detach method disconnects the subscribe channel that is referenced by the $id parameter. The first channel is used if no parameter is supplied.

Parameters

Integer	*id*	Optional session ID.

Returns

Always returns '1' on success; internal exception thrown on failure.

dispatch

Signature

```
dispatch($timeout_ms[, $breakOnEvent]);
```

Description

This function loops while waiting for messages from the Hub. A timeout function is run if it has been set. Reconnects to the Hub in the event this connection drops.

Parameters

Integer	*Timeout_ms*	Time to wait for a new message on a session
Integer	*breakOnEvent*	Stop after a message has been received.

Returns

Session error code (NIMSW_TIMEOUT, NIMSW_ERROR, NIMSW_RESTART or NIMSW_EXIT) or -1 on a fatal error.

server

Signature

```
server([$port[, $timeoutCB[, $restartCB]]]);
```

Description

Starts a server session, which waits for incoming messages on the Nimsoft Bus. Callbacks which have been registered are published on the bus, making the interface available.

Parameters

Integer	*port*	Optional port on which to listen for incoming connections.
Function	*timeoutCB*	Optional Function to run on timeout.
Function	*restartCB*	Optional Function to run on restart.

Returns

iRet - The return code. Returns NIME_OK on success, or a NIME_{xxx} error code on failure, where {xxx} is the specific error code (see page 9).

setInfo

Signature

```
setInfo($version[, $company]);
```

Description

Fills in the information about the probe which is gathered when the **_status** callback is run.

Parameters

String	*version*	Version information.
String	*company*	Company which wrote the probe.

Returns

iRet - The return code. Returns NIME_OK on success, or a NIME_{xxx} error code on failure, where {xxx} is the specific error code (see page 9).

setPostCallback

Signature

```
setPostCallback($function_name);
```

Description

Sets the function to run when a message is received from the Hub. Default function is **hubpost**, but this allows you to specify another function name.

Parameters

Function	*function_name*	Function to run when a message is received from the Hub.

Returns

iRet - The return code. Returns NIME_OK on success, or a NIME_{xxx} error code on failure, where {xxx} is the specific error code (see page 9).

setRetryInterval

Signature

```
setRetryInterval($intervalAsSeconds);
```

Description

Sets how often to attempt a reconnect if connection to the Hub drops.

Parameters

Integer	*intervalAsSeconds*	Time in seconds between reconnection attempts.

Returns

iRet - The return code. Returns NIME_OK on success, or a NIME_{xxx} error code on failure, where {xxx} is the specific error code (see page 9).

subscribe

Signature

```
subscribe($subjects[, hubip[, hubport]]);
```

Description

The subscribe method causes the session to set up a subscribe channel at the hub. This type of channel is not secure in the sense of deliverability. Messages are only directed down the channel when the session is listening.

Parameters

String	*subject*	Comma separated list of subjects to listen for on the Hub
String	*hubip*	Optional IP address of the Hub
Integer	*hubport*	Optional Port on which the Hub is listening (48002)

Returns

iRet - The return code. Returns NIME_OK on success, or a NIME_{xxx} error code on failure, where {xxx} is the specific error code (see page 9).

See Also

attach (see page 52)

Chapter 3: Examples for Developing Solutions using Perl

This section contains the following topics:

Retrieving Data from the Configuration File

All probes (or standalone programs) must have be allowed to get configuration data during startup/initialization. A probe running under a Nimsoft Robot is part of a fully distributed system. This means that the configuration file may be edited using a specific configuration-tool, or a generic tool that is provided by Infrastructure Manager.

Example

```
use Getopt::Std;
use Nimbus::CFG;# Reference to Nimsoft Config module
use strict;

my %options;
getopts("d:m::",\%options);
############################################################
# Read configuration data into config. object and set the directory and
# max values with the data according to the following precedence
#    1. Command-Line
#    2. From configuration file
#    3. Default values (hard coded)
#
my $cfg = Nimbus::CFG->new("filemonitor.cfg");

my $dir = $options{d} || $cfg->{"setup"}->{"directory"} || ".";
my $max = $options{m} || $cfg->{"setup"}->{"max_files"} || 10;
```

Building and Publishing a User-defined Message

The Perl module Nimbus::PDS wraps the various PDS functions in the Nimsoft API function library.

Example

The example creates a PDS instance, adds some data to the PDS, then publishes the message onto the Nimsoft Bus with the subject "your_subject".

```
use Nimbus::PDS
#############################################################
# Post trend-data onto the Nimsoft Bus
#
my $pds    = Nimbus::PDS->new();
my $dir    = "/tmp";
my $count  = 33;

$pds->string ("directory",$dir);
$pds->number ("count",$count);
$pds->post   ("your_subject");
```

Sending an Alarm Message

Sending a Nimsoft Alarm message can assist in the integration of existing scripts with event management, or when you choose to use Perl as the programming language for system administrative tasks (for example, monitoring and batch jobs).

Synopsis

```
use Nimbus::API;

my($retcode,$msgid) = =nimAlarm (see page 13) ($severity[,$msgtext [,$subsysid
[,$supid[,$source]]]);
```

where:

- Severity and msgtext are required parameters, while the others are optional. The severity parameter may be one of the severity level constants (see page 13) or its corresponding number.
- The subsysid may be a registered subsystem identification number stored in the Nimsoft Alarm Service (nas), or it may be any string.
- The supid may be used to group/tag one or several events into a state. The state may be used to notify the operator about a situation, yet remove the event from the console whenever the situation is back to normal again.
- The source may be used to impersonate any equipment so it appears as a managed element.

Examples

A simple example showing how to send an alarm.

```
use Nimbus::API;
nimAlarm(NIML_CRITICAL, "This is a critical message from perl");
```

This next example is a bit more complicated showing how to integrate a simple file counting module with Nimsoft, that should give an alarm when the number of files in the /tmp directory exceeds 3, and clears it if the number of files is 3 or less.

```
##########################################################################
# Send an alarm with a suppression identification tag
##########################################################################

use Nimbus::API; # Reference to Nimsoft Programming Interface
use Monitor;       # Reference to module containing FileCount
use strict;

my $dir = "/tmp";

##########################################################################
# The subsystem-id is maintained by the Alarm Server, you may however,
# also set it to any textstring like "DirMonitor". Here we use the id.
# Hint: launch the NAS configurator and find the string tied to the id by
# looking under the subsystem tab.
#
my $sid = "1.1.5";

##########################################################################
# The suppression tag enables the Alarm Server to keep a severity state on
# the message so it is possible to escalate the severity and/or to clear it.
#
my $suptag = nimSuppToStr(0,0,0,"fileMonitor/$dir");

my $count = FileCount($dir);

if ($count > 3) {
        nimAlarm(NIML_CRITICAL,"Directory $dir has too many files
($count)",$sid,$suptag);
}else {
        nimAlarm(NIML_CLEAR,"Directory $dir is checked and ok",$sid,$suptag);
}
```

Building a Server Solution

You may, by simple means, make your Perl script behave as a "Network-aware application." It can be upgraded from just a monitoring probe, generating alarms in certain situations, to a full-blown server responding to requests over the network. The Nimbus::Session module is a class module wrapping many features from the C-like interfaces found in Nimbus::API. The session object decreases the number of API functions needed to a minimum.

Steps in creating your Perl server:

1. Insert references to Nimbus::API and Nimbus::Session
2. Construct a new session object using Nimbus::Session->new
3. Initiate a server using Nimbus::Session->server
4. Register callback functions using Nimbus::Session->addCallback
5. Invoke the message dispatcher using Nimbus::Session->dispatch.

Example

```
use Nimbus::API;
use Nimbus::Session;
use strict;

#############################################################
# CALLBACK Declarations
sub your_function {
        my ($hMsg,$str_param,$int_param) = @_;
        print "your_function: I received a string=$str_param, and a number=$int_param\n";
        nimSendReply ($hMsg);
}
sub timeout {
        # Do something, normally you would put your monitoring code
        # here, checking the elapsed time.
}
sub restart {
        # Reload configuration, etc...
}
```

```
############################################################
# MAIN ENTRY
############################################################
my $sess = Nimbus::Session->new("perl-example-server");
$sess->setInfo ("1.0", "Owner information goes here");

if ($sess->server (NIMPORT_ANY,\&timeout,\&restart)==0) {
        $sess->addCallback ("your_function", "string_param, integer_param%d" );
        $sess->dispatch ();
}
```

Subscribing to Bus Messages

Any message published onto the Nimsoft Bus may be subscribed to by a program or script. The Nimbus::API provides the generic functions for subscribing to a message. The simplest way is to use the Nimbus::Session module. This module implements a method called Nimbus::Session->subscribe.

Example

```
use Nimbus::API;
use Nimbus::Session;
use strict;

############################################################
# Callback whenever a message is ready for delivery, the udata
# and full messages are PDS containing the full (complete) message
# including headers, and the udata is the user-data area of the message.
sub hubpost {
   my ($hMsg,$udata,$full) = @_;

   nimLog (1,"(hubpost)");
   my $subject = pdsGet_PCH($full,"subject");
   print "The user-data posted under subject: $subject\n";
   pdsDump($udata);

   nimSendReply($hMsg);
}
```

```
############################################################
# Other CALLBACK Declarations
sub your_function {
        my ($hMsg,$str_param,$int_param) = @_;
        print "your_function: I received a string=$str_param, and a number=$int_param\n";
        nimSendReply ($hMsg);
}
sub timeout {
        # Do something useful
}
sub restart {
        # Reload configuration, etc...
}
sub ctrlc {
   exit;
}
############################################################################
# Main entry
############################################################################
$SIG{INT} = \&ctrlc;

nimLogSet ("stdout","test",1,0);
nimLog    (0,"Starting..");

my $sess = Nimbus::Session->new("perl-example-server");

$sess->setInfo ("1.0", "Owner information goes here");

if ($sess->subscribe ("MY_SUBJECT")) {
   nimLog(0,"unable to subscribe at Hub");
}
if ($sess->server (NIMPORT_ANY,\&timeout,\&restart)==0) {
        $sess->addCallback ("your_function", "string_param, integer_param%d" );
        $sess->dispatch ();
}
nimLog (0,"Exiting...");
```

Troubleshooting

Symptom: Scripts halt with an error message similar to:

```
Can't locate Nimbus/API.pm in @INC (@INC contains:
/usr/local/lib/perl5/sun4-solaris/5.00404 /usr/local/lib/perl5
/usr/local/lib/perl5/site_perl/sun4-solaris /usr/local/lib/perl5/site_perl .) at
subex.pl line 1.
BEGIN failed--compilation aborted at subex.pl line 1.
```

Possible causes:

1. The perllib directory structure is not reachable for Perl
   - Add PERL5LIB to your environment (for example PERL5LIB = /opt/nimsoft/perllib)
   - Add **use lib** ("/opt/nimsoft/perllib") to your script header (before **use Nimbus::API**)
2. Perl Extensions for Nimsoft are binary incompatible with the distribution
   - Get a compatible Perl distribution (or contact Nimsoft support).

Symptom: You are unable to locate the perllib directory on the file system.

Possible causes:

1. The Nimsoft runtime libraries for Perl have not been installed
   - Install the runtime libraries
2. They do not reside under the root installation directory for Nimsoft
   - WIN32: c:\program files\Nimsoft\perllib
   - UNIX: /opt/nimsoft/perllib.

Chapter 4: Programming in the SLM Environment with Perl

This section contains the following topics:

Header Files and Libraries

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

```
use Nimbus::API;
```

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. This means you must send a QOS_DEFINITION message. Providing the nimQoSSendDefinition function with a correct and valid parameter list generates output similar to the following:

```
nimQoSSendDefinition ("QOS_TEST",                  # QOS Name
                      "QOS_APPLICATION",           # QOS Group
                      "Test Application Response", # QOS Description
                      "Milliseconds","ms",0,0);    # Unit info and flags.
```

Sending QoS Data

The probe should only initialize itself during startup by sending the 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:

```
my $samplevalue = 100;
my $source      = nimGetVarStr(NIMV_ROBOTNAME);
my $target      = "startup-time";
my $interval    = 300;                                  # Seconds

if (my $qos = nimQoSCreate("QOS_TEST",$source,$interval,-1)) {
        if ($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. This is useful when doing response time measurements:

```
my $source     = nimGetVarStr(NIMV_ROBOTNAME);
my $target     = "startup-time";
my $interval   = 300;                                  # Seconds

if (my $qos = nimQoSCreate("QOS_TEST",$source,$interval,-1)) {
        nimQoSStart($qos);

        # Do your work here...

        nimQoSStop($qos);
        nimQoSSendTimer($qos,$target);
        nimQoSFree($qos);
}
```