NimBUS Script Agent (nsa)

Technical Brief

Version: 2.06
Date: March 18, 2015
Author: cseeberg

Table of Contents

GENERAL **3**

TECHNICAL OVERVIEW **3**

MySQL support 3

CONFIGURATION AND DATA FILES **3**

CLASS DEFINITIONS **4**

Database *4*
Action *4*
Nimbus *4*
File *5*
Timestamp *6*
Probe *7*
Net *8*
SNMP *9*
PDS *10*
DLL *11*
Language Extension *12*

TROUBLESHOOTING **14**

Unable to access embedded class functions 14

APPENDIX **15**

The NSA Debugger 15

General

The NimBUS Script Agent (nsa) is a Lua-based scripting platform. The nsa is meant to be used as a tool for system integration, reporting, on-site enhancements allowing for direct integration with NimBUS, ADO, MySQL, SNMP and basic system and network functions. It requires almost no run-time environment, not even NimBUS and contains all functionality within a single binary. The user may encrypt sensitive data (like NimBUS or database login passwords) and incorporate this into the script(s).

Technical Overview

The nsa is written in C and built using the latest libraries like Lua 5.1, net-snmp 5.4.1, SQLite 3.5.4 and NimBUS 4.x. The windows binary also supports ADO allowing for direct access to the myriad of database providers.

The nsa can be run in a command shell window as a manual management/reporting tool, or it can be used as a timed or daemon probe running under the NimBUS robot.

Scripts can be written in any editor, a great editor of choice is “the crimson editor” - http://www.crimsoneditor.com/ with built-in lua support and ways to extend the syntax highlighter with the nsa extensions. In addition it has capabilities of executing the scripts and grabbing the output thus becoming an Integrated Development Environment (IDE).

MySQL support

NSA 1.14 and later will attempt to dynamically load libmysql.dll / libmysql.so in order to provide MySQL support. You will need a MySQL connector for C on your system if you wish to access MySQL database through NSA.

Configuration and Data Files

The nsa is configured by a standard set of command-line options,

-	a <argument>	passes argument string into script.
-	l <logfile>	sets the logfile e.g stdout. (default:stdout or <probename>.log)
-	d <debug level>	defines which loglevel the script should run under
-	p <string>	creates an encrypted string based on *string*.
-	P <string>	creates a host-locked encrypted string.
-	D	starts NSA in debugger mode.

The following environment variables will be honored by the NSA:

NIM_ROOT - this is typically set by the NimBUS robot when running as a probe.
NIM_LUA_PATH - this variable will be pre-pended to the script lookup path.
NIM_LUA_CPATH - this variable will be pre-pended to the module lookup path.
LUA_PATH - will override all script lookup paths.
LUA_CPATH - will override all module lookup paths.

NSA will use the environment variables to look for scripts and modules in a manner specified by the lookup path when using the **require** function.

Class Definitions

Database

database.open ([sFileName | sConnectionString [,bStopOnError]])

Opens a database handle to the specified file or database. Subsequent *database* operations will now be reference through this handle, until it is closed using the *database.close* or through an implisit close when opening another database using *database.open.* The default database is called *user.db*. The *ConnectionString* parameter may also be an encrypted string using the –p/-P command-line options. This function returns an error code (see Constants) when *StopOnError* is false. Default is to abort on error.

E.g:

database.open ("myprivate.db") or
database.open ("Provider=SQLOLEDB;Initial Catalog=NimbusSLM;Data Source=myserver; User ID=sa;Password=mypassword;Network Library=dbmssocn;Language=us_english")

database.query (sSQL)

Performs the provided SQL in the current open database. If no previous *database.open* has been performed then the *user.db* is used. The SQL statement must be supported by the underlying database.

database.close ( )

Closes the current database.

database.setvariable (sName, sValue)

Creates (or modifies) the persistent variable *Name* in the current database. The variable name should be a unique name to avoid collisions.

database.getvariable (sName)

Retrieves the persistent variable *Name.* The function returns the value as a string as well as the optional modification timestamp, *nil* when the variable is non-existent.

Action

action.command (sCommandLine)

Executes the provided command-line string, and places the output (if any) into a table of lines. The exit-code is returned as the second output parameter. E.g. output, rc = action.command ("ls –al")

action.ping (sHostName [, iTimeout])

Returns the status (true or false) and the time-used (in milliseconds) when issuing a ping (ICMP ECHO) to the provided hostname or ip-address.

action.email (sReceiverAddress, sSubject [, sBody]])

Generates an email-message targeted for the NimBUS Email Gateway. Returns true when successful.

action.SMS (sPhoneNumber, sMessageText)

Generates an SMSl-message targeted for the NimBUS SMS Gateway. Returns true when successful.

Nimbus

nimbus.login (sUsername, sPassword)

Login to NimBUS with the provided username and password. Note that the password can be encrypted using the –p/-P command-line arguments and that it is possible to utilize the probe security in the probe package when running as a probe (using the probe class). Returns the result as boolean, and the session identification as string.

nimbus.log (iLogLevel, sFormatString, args…)

Writes a line to the specified logfile like the sprint function. The configured loglevel will determine if the logline is written. E.g configured loglevel is 1.

```
nimbus.log (0,"This will be written")
nimbus.log (1,"This will also be written")
nimbus.log (2,"This will not be written")
```

nimbus.setloglevel (iLogLevel)
Sets the current loglevel to *LogLevel*.

nimbus.alarm (iSeverityLevel, sMessageText [, sSuppressionKey [, sSubsystemId [, sSource]]])
Generates a NimBUS alarm message with the severity level (1-5) and a message-text. Use the suppression-key to create a stateful alarm. Returns a return code and the message-id string.
E.g. rc,nimid = nimbus.alarm (NIML_WARNING, "help me..")

nimbus.post (sSubject, PDSHandle)
Posts a NimBUS Message onto the NimBUS using the Subject.
Returns a message-id string if successful or **nil**.

nimbus.request (sNimBUSAddress, sCommand [, sArguments [, iWait [, ReturnAsPDS]]])
Returns the result of the command targeted for the provided nimbus component. The command-arguments are expected to be a PDS (returned by pds.create). The result is placed into a table unless the ReturnAsPDS parameter is set to *true*.
Please note that this is an associative table (not indexed), meaning that a PDS sections will be referenced by its section-name.

```
controller = nimbus.request ("controller","get_info")
printf ("controller robot: %s", controller.robotname)
```

nimbus.qos_definition (s QosName, sQosGroup, sDescription, sUnit, sUnitAbbreviation, bHasMax [, bIsAsynch])
Creates a QoS definition named *QosName*. Unless the flag IsAsynch is *true*, an interval based QoS is created. Please note that subsequent definitions on the same name will not recreate or alter an existing QoS definition. The HasMax flag set requires that all qos data (issued by *nimbus.qos*) referring to this *QoSName* is issued with a MaxValue.

nimbus.qos (sQosName, sSource, sTarget, dValue, iInterval | QOS_ASYNCH [,nMaxValue])
Will send an interval based QoS message when *Interval* is greater than zero, and a asynchronous QoS message when called with QOS_ASYNCH. Please note that no QoS data will be recorded unless a valid QoS definition has been sent prior to this request. Remember to set the MaxValue if definition was created using HasMax=true.

nimbus.session_open (sNimBUSAddress)
Opens a session to the targeted NimBUS component. Returns a handle to the session.

nimbus.session_request (pSessionHandle, sCommand [, sArguments [, iWait [, iReturnAsPDS]]])
See nimbus.request.

nimbus.session_close (pSessionHandle)
Closes and removes the data structure associated with the handle.

nimbus.setsid(sSid)
Sets the session identification information into the current running environment.

nimbus.encrypt (sString, sSecretKey)
Returns a base64 encoded string using twofish encryption.

nimbus.decrypt (sString, sSecretKey)
Returns the decoded string using the key.

File

file.copy (sSource, sDestination)
Creates a file using the complete *Path* and writes *Buffer* into the file if provided.

file.create (sPath [, sBuffer])
Creates a file using the complete *Path* and writes *Buffer* into the file if provided.

file.delete (sPath)
Deletes the file named *Path.*

file.read (sPath [,sMode [,iStartPos]])
Returns a buffer with the filecontents, and the number of bytes read as a second return parameter. The optional *mode* parameter allows for controlling the open-mode. (see fopen man-pages, default: "rb"), *StartPos* will indicate where the reading should start (default: 0)

file.write (sPath , sBuffer)
Appends *Buffer* the file *Path*, and returns **true** if success

file.stat (sPath)
Returns a table containing the following statistics: *mtime, ctime, atime, mode* and *size*.

file.rename (sOldName , sNewName)
Renames the file OldName to NewName.

file.checksum (sPath)
Returns a Base64 encoded checksum string for the specified file.

file.list (sPath [, sPattern [, bDirectoriesOnly]])
Returns a string table with the filenames (or directories if the DirectoriesOnly is set to **true**). The *pattern* can be used to specify the search pattern (default is *).

Timestamp

timestamp.now ()
Returns the number of seconds elapsed since Jan. 1 1970, 00:00:00.

timestamp.diff (iStartTimeStamp [, sFormat [, iEndTimeStamp]])
Returns the difference (seconds, minutes,hours or days) between the EndTimeStamp (or *now* if not provided) and the StartTimeStamp using the Format specifier (**s**econds, **m**inutes, **h**ours,**d**ay)

timestamp.newer (iTimeStamp, sTimeSpecification)
Returns **true** if the TimeStamp is newer than specified by the TimeSpecification. The TimeSpecification format is built using a combination of numbers and the tokens: **s**econds, **m**inutes, **h**ours, **d**ays. E.g. 10h30min, 5hrs, 30m, 3 days

timestamp.older (iTimeStamp, sTimeSpecification)
Returns **true** if the TimeStamp is older than specified by the TimeSpecification. The TimeSpecification format is built using a combination of numbers and the tokens: **s**econds, **m**inutes, **h**ours, **d**ays. E.g. 10h30min, 5hrs, 30m, 3 days

timestamp.data ([iTimeStamp])
Uses '*now'* if no parameter is provided. Returns a table with the following self-explanatory members: *year,month,day,hour,minute,second,yearofday,weekday* and *isdst* (1 if daylight savings time).

timestamp.fromISO (sISOdatestring)
Returns a timestamp and a timestamp data table (see *timestamp.data*).

timestamp.format (iTimeStamp [, sFormat])
Returns a formatted timestring using the Format specifier (default: %b %d, %H:%M:%S).

specifier	Replaced by	Example
%a	Abbreviated weekday name *	Thu
%A	Full weekday name *	Thursday
%b	Abbreviated month name *	Aug
%B	Full month name *	August
%c	Date and time representation *	Thu Aug 23 14:55:02

		2001
%d	Day of the month (01-31)	23
%H	Hour in 24h format (00-23)	14
%I	Hour in 12h format (01-12)	02
%j	Day of the year (001-366)	235
%m	Month as a decimal number (01-12)	08
%M	Minute (00-59)	55
%p	AM or PM designation	PM
%S	Second (00-61)	02
%U	Week number with the first Sunday as the first day of week one (00-53)	33
%w	Weekday as a decimal number with Sunday as 0 (0-6)	4
%W	Week number with the first Monday as the first day of week one (00-53)	34
%x	Date representation *	08/23/01
%X	Time representation *	14:55:02
%y	Year, last two digits (00-99)	01
%Y	Year	2001
%Z	Timezone name or abbreviation	CDT
%%	A % sign	%

* The specifiers whose description is marked with an asterisk (*) are locale-dependent.

Probe

The Probe class allows the script programmer to write a full-blown NimBUS probe in the Lua language like you can using C/C++, Java, Perl, VB, COM etc.

probe. register(sName, sVersion, sCompanyCopyright [,sLogFile [,iLogLevel [,iLogFlags]]])
Registers the named probe with the NimBUS Robot. Note that this method is prerequisite to all of the following methods. The Version string is on the form <major>.<minor><release> e.g. 2.01,.
The default *LogFile* is <*Name*>.log

probe. unregister()
Unregisters the probe.

probe. run([iTimeout])
This suspends further execution until a "stop" command is issued to the probe. User callbacks and standard callbacks like *timeout, restart* and *stop* will be dispatched when received. The Timeout parameter indicates how often the *timeout* callback-function is called. Default is 5000ms.

probe. dispatch([iTimeout])
This method returns control to the script with a return code indicating which event was dispatched. The registered and standard callbacks are fired as with probe.run(). This gives the script programmer full control of the event dispatcher.

probe. addCallback(sCallbackFunctionName, sCallbackArguments [, iSecLevel])
This will add a callback function to the probe dispatching mechanism. The CallbackFunctionName is a string with the same name of an existing function. The CallbackArguments are used to inform the callers of the parameters and the data-type available. Use SecLevel to alter the security level of the callback. The default value is 0 (Open).

E.g. probe.addCallback("get_info","details%d") will inform the callers that the "get_info" command takes an integer argument named details. The supported identifiers are %d and %s (for string). String is assumed if no identifier is used.

probe. subscribe(sSubject [,sCallbackFunctionName [,sHubAddress]])
This will open a subscriber channel to the hub denoted by the HubAddress (default HubAddress="hub") with a subject list specified by Subject. The default CallbackFunctionName is "hubpost". The callback function synopsis is *hubpost (messagedata [, msgheaderdata, [, PDSmessagedata, [, PDSmsgheader]]])*
The subscribe channel is maintained (reconnected when disconnected) by the nsa.

probe.attach (sQueueName [, sCallbackFunctionName [, sHubAddress]])
This openes a subscriber channel to a named queue on the target HubAddress. See probe.subscribe()

probe.config ([sConfigFile])
Reads the configuration file. Default configuration file is <Name>.cfg where Name is the registered probe name. The configuration file is returned as a table, where the full section path name is used as the key for the section.

```
local cfg = probe.config()

if cfg ~= nil then
    -- Look for spesific key/value pair.
    if cfg["/setup"] then
        probe.log(0,"Loglevel is %d",cfg["/setup"].loglevel)
    end
    -- Look for a subsections e.g profiles where active = yes is a keyvalue pair.
    for k,p in pairs(cfg) do
        if substr(k,"/profiles/") then
            probe.log(0,"%s active: %s",k,p.active)
        end
    end
end
```

probe. log(iLogLevel, sFormat [, Arguments])
See nimbus.log()

probe. setloglevel (iLogLevel)
See nimbus.log()

Net

The Net class provides a few simple but useful networking methods.

net.ping (sHostname | sIP-address [,iTimeout [,iNumPackets [,iPacketSize]]])
Send ping (ICMP ECHO) to the designated host. Timeout is in ms, default 1000.
Returns status (true or false) and time used in milliseconds.

net. connect(sHostname | sIP-Address, iTCPport [, iTimeout])
Attempts a TCP connect to the designated host. Timeout is in ms, default 1000.
Returns status (true or false) and time used in milliseconds.

net. nametoip(sHostname)
Returns the IP-address associated with the Hostname.

net. iptoname(sIp-Address)
Returns the hostname (if any) associated with the ip-address.

net. gethostname()
Returns the hostname of "this" system.

SNMP

The SNMP class provides the script programmer with built-in SNMPGET and SNMPWALK functionality.

snmp.create (iSnmpVersion, sTargetHost, …)

The snmp.create method takes different arguments dependent on the SNMPversion information.

For version 1 and 2:
snmp.create(1|2,target,community [,oidlist [,options]])

For version 3:
snmp.create (3,target,username,password,auth,seclevel [,privprotocol [,privpassword [,oidlist]]])

This method returns a handle used by the other methods in this class. The oidlist is a comma or white-space separated list of object identifiers. They will be added to the handle's request list, and used by the snmp.query method.

snmp. addvariable (pSnmpHandle , Oid)

Adds the oid to the SnmpHandle request buffer. Will be used by *snmp.query*.

snmp. addvariable (pSnmpHandle , sFormat, sType, sData [, iIndex])

Adds the oid to the SnmpHandle set buffer. The Format string is a sprint like formatting string allowing you to modify the OIDs used together with the snmp.set. The Type is one of *string, number* or *time.* This usage of the addvariable method will be only used by *snmp.set*.

```
h = snmp.create (1,"193.71.55.249","mysecret")

-- Add variables to the handle's request oidlist
snmp.addvariable (h,".1.3.6.1.2.1.1.4.0","string","New Contact")
-- Or by using the index parmeter
snmp.addvariable (h,".1.3.6.1.2.1.1.4.%d","string","New Contact",0)

-- Perform the actual SNMP set
snmp.set (h)
snmp.delete(h)
```

snmp. delete(pSnmpHandle)

Removes the data-structure associated with SnmpHandle.

snmp.query (pSnmpHandle)

Runs the query against the oidlist specified in the create statement.

snmp. get(pSnmpHandle,Oid)

Requests the Oid from the targethost denoted by the SnmpHandle. The data is returned as a table.

snmp. set(pSnmpHandle)

Sets the variables added to the handle by *snmp.*addvariable at the targethost denoted by the SnmpHandle. The data is also returned as a table if successful as well as a return code.

snmp.getnumber (tTable|pSnmpHandle, sOid)

Requests a single integer value from the SnmpHandle's result buffer, or from the Table returned by snmp.walk or snmp.get.

snmp.getstring (tTable|pSnmpHandle, sOid)

Requests a single string value from the from the SnmpHandle's result buffer, or from the Table returned by snmp.walk or snmp.get.

snmp. walk (pSnmpHandle, sStartingOID, sRootPath [, iMaxOids])

Performs a "snmpwalk" starting at the StartingOID, limiting it to the RootPath. The optional MaxOids parameter allows the caller to set the maximum oids returned per request. The default value is 300. The table returned by this function contains the following elements: **numoids, done, lastoid** and **oids.**

```
h   = snmp.create (1,"193.71.55.245","public")
out = snmp.walk   (h,"1.3.6.1.2.1","1.3.6.1.2.1")
idx = 0

while out ~= nil do

    for i=0,out.numoids-1 do
        printf ("%d oid: %s, type: %s", idx, out.oids[tostring(i)].oid, out.oids[tostring(i)].type)
        idx=idx+1
    end

    if out.done == 1 then break end

    out = snmp.walk   (h,out.lastoid,"1.3.6.1.2.1")
end
snmp.delete(h)
```

PDS

The PDS (Portable Data Stream) format is used heavily within the NimBUS to exchange data between various processes on all platforms supported by NimBUS. This format allows users to build nested datastructures that may be passed between different languages and different hardware platforms.

pds.create ()

Returns a reference handle to a PDS structure. Use **pds.size** (pdsHandle) to obtain the size of the PDS.

pds.copy(pdsHandle)

Returns a reference handle to a copied PDS structure.

pds.delete (pdsHandle)

Deletes the PDS structure and data.

pds.convert (pdsHandle)

Returns a LUA table. This function converts the PDS structure to a LUA table containing the same key/value pairs and sub-tables (if any).

pds.putInt (pdsHandle, sKey, iValue)

Stores an integer value in the provided PDS structure using the Key as the reference to the Value. Note that an existing element with the same Key will be replaced.

pds.putString (pdsHandle, sKey, sValue)

Stores a string in the provided PDS structure using the Key as the reference to the Value. Note that an existing element with the same Key will be replaced.

pds.putDouble (pdsHandle, sKey, dValue)

Stores a double value in the provided PDS structure using the Key as the reference to the Value. Note that an existing element with the same Key will be replaced.

pds.putPDS (pdsHandle, sKey, pdsHandle)

Stores a PDS in the provided PDS structure using the Key as the reference to the Value. Note that an existing element with the same Key will be replaced.

pds.putTable (pdsHandle, sKey, Value)

Stores a *Value* of type *string, number* or *PDS* to the named table *Key*.

pds.getInt (pdsHandle, sKey)

Returns the number associated by Key from the provided PDS structure (or **nil** if non-existent).

pds.getString (pdsHandle, sKey)

Returns the string value associated by Key from the provided PDS structure (or **nil** if non-existent).

pds.getDouble(pdsHandle, sKey)
Returns the number associated by Key from the provided PDS structure (or **nil** if non-existent).

pds.getPDS (pdsHandle, sKey)
Returns the PDS handle associated by Key from the provided PDS structure (or **nil** if non-existent).

pds.getTableInt (pdsHandle, sKey, iTableIndex)
Returns the number associated by Key from the named table Key and index *TableIndex*. Zero (0) is the first table index.

pds.getTableString (pdsHandle, sKey, iTableIndex)
Returns the string value associated by Key from the named table Key and index *TableIndex*. Zero (0) is the first table index.

pds.getTablePDS (pdsHandle, sKey, iTableIndex)
Returns the PDS handle associated by Key from the named table Key and index *TableIndex*. Zero (0) is the first table index.

pds.getNext (pdsHandle)
Returns the next Key, Type, DataSize, Data from the provided PDS structure (or **nil** if non-existent).

pds.fileOpen (sPath)
Returns a reference handle to an open pdsFile. Close the file using **pds.fileClose**.

pds.fileClose (pdsFileHandle)
Closes the pdsFile.

pds.fileRead (pdsFileHande [,bMarkAsRead])
Returns 3 optional output parameters: return code, a reference handle to a PDS and the number of bytes read. The bMarkAsRead flag advances and saves the file read pointer, default is *true*. Note that the PDS file can contain blocks of PDS's, these can be read in a loop.
E.g `rc,dta,nbytes = pds.fileRead(f,true)`

pds.fileWrite (pdsFileHande , pdsHandle)
Writes the provided pdsHandle data to the file.

DLL

dll.load (sFilename)
Returns a handle to the opened dynamic library.

dll.free (dllHandle)
Free's up the data-structures related to the dll.

dll.call (dllHandle, sFunctionName [, args])
Calls the dll-function *sFunctionName* with up to 10 arguments. Make sure that the argument count matches the expected arguments for the dll function. This function supports lua types: number and string only.

dll.icall (dllHandle, sFunctionName [, args])
Expects a returned integer value from the dll-function and passes this back to the caller as a string.

dll.scall (dllHandle, sFunctionName [, args])
Expects a returned string value from the dll-function and passes this back to the caller as a string.

```
h = dll.load ("winmm.dll")
dll.call     (h, "PlaySoundA","C:\\WINDOWS\\MEDIA\\notify.wav", 0, 0)
dll.free     (h)
```

sprintf (sFormat [,Par1 [,Par2 […]]])
Returns a string buffer with the formatted string.

printf (sFormat [,Par1 [,Par2 […]]])
Logs the formatted string to the output window (if in the editor) or the NAS logfile.

print (Par1[,Par2 […]]])
Logs the string to the output window (if in the editor) or the NAS logfile. Used primarily for simple unformatted printing and debug output.

left (sString, iLength)
Returns *Length* characters from the *String*, starting from the left.

right (sString, iLength)
Returns *Length* characters from the *String*, starting from the right.

mid (sString, iStart [, iLength])
Returns *Length* characters from the *String*, starting from *Start*. If no *Length* is specified, the rest of the string will be returned.

substr (sString, sSubstring)
Returns **true** if the *Substring* is found, as well as the starting *Position* of the subsctring.

split (sString [, sSeparators])
Returns a table of substrings separated by one or more of the Separator characters. The default separator is whitespace.

trim (sString [, iMode])
Returns a *String* trimmed for leading and/or trailing whitespaces.
The *Mode* is 0 – leading and trailing, 1 – leading only and 2 – trailing only

regexp (sString, sExpression)
Returns **true** if the regular (or pattern matching) expression matches *String*.

setvariable (sName, sValue)
Stores the non-persistent variable *Name*. The value is retrievable until a cold-start of the NSA clears the Non-persistent data store. Use the equivalent **database.setvariable** for a persistent store. NSA clears the
variable if sValue is *nil*.

getvariable (sName)
Returns the non-persistent named variable *Name* or **nil** if non-existent.

exit (iExitCode)
Terminates the script execution with an *ExitCode*. Non-zero *ExitCodes* will be recorded in the NAS activity-log.

tonumber (Value)
Converts *Value* into a number.

tostring (Value)
Converts *Value* into a string.

type (Value)
Returns the variable type as as string.

sleep (iMilliSeconds)
Suspends execution for a given time.

setenv (sName, sValue)
Sets the environment *Name* to *Value.*

thread (LuaCodeblock [,Argument])

Executes the *LuaCodeBlock* in a separate thread, that allows for "parallel" execution outside the current script context. The lua interpreter already contains co-routines that allows for controlled processing. The variable THREAD_ARGUMENT is filled with the optional parameter *Argument* and is used to pass data from the calling script. Please note that this inline code runs in a complete separate context, thus not able to access any data in the callers context. The *LuaCodeBlock* may be a quoted string, or a string literal enclosed in double square bracket. [[..]]

```
-- Inline thread code
function worker()

    thread ([[
        var = split(THREAD_ARGUMENT,",")
        probe.log (0,"var is '%s'",var[1])

        for i=1,10 do
            probe.log(0,'thread working')
            sleep(1000)
        end
        probe.log(0,"thread done...")
        ]],"arg1,arg2")

end
```

Constants

Script constants:

SCRIPT_FILE	= filename of script.
SCRIPT_ARGUMENT	= argument string passed by the –a command-line switch.

Alarm severity levels:

NIML_CLEAR	= 0
NIML_INFORMATION	= 1
NIML_WARNING	= 2
NIML_MINOR	= 3
NIML_MAJOR	= 4
NIML_CRITICAL	= 5

Error codes:

NIME_OK	= Ok
NIME_AGAIN	= Not ready, try again
NIME_ERROR	= Error
NIME_COMERR	= Communication/connectivity error
NIME_INVAL	= Invalid argument
NIME_NOENT	= No such entry
NIME_ISENT	= Entry is already defined
NIME_ACCESS	= No access

Probe dispatcher return codes:

NIMSW_TIMEOUT	= Timeout value is reached
NIMSW_MSG	=
NIMSW_ERROR	= Error situation is detected, e.g. disconnected subscriber channel.
NIMSW_EXIT	= The “stop” command has been issued.
NIMSW_RESTART	= The “restart” command has been issued.
NIMSW_SHUTDOWN	= A “shutdown” command has been issued

Troubleshooting

Unable to access embedded class functions

Problem description

Error message is displayed when accessing a built-in function like database.open.
E.g. Jul 6 12:54:21:756 nsa: Script error, test.lua:12: attempt to call field 'open' (a nil value)

Solution

This problem will occur when you override/overload the class name (in this case, database) with a local variable. The solution is to rename your variable.

Appendix

The NSA Debugger

The built-in NSA debugger (from version 2.00) allows the script developer to perform common debugging functions like setting breakpoints, stepping in/out of functions, showing data etc.
It is invoked using the –D option. The following command-set is supported:

```
%nsa –D script.lua

====== NSA 2.00 DEBUGGER STARTING =======
nsa>?
Available Commands:
  r|run                                 execute script.
  c|cont                                continue execution.
  n|next                                next statement, step over functions.
  s|step                                step into functions.
  e|exit                                exit NSA.
  q|quit                                quit debugging, and reload script.
  l|list [num_lines]                    list num_lines of script sourcecode.
  p|print [variable|*]                  print all or named variables.
  b|break [file:]lineno|funct.      set breakpoint.
  t|bt|backtrace                        show stacktrace.
  B|Breakpoints                         list breakpoints
  clear [file:]lineno|funct.            clear spesified breakpoint.
  where                                 print current line.

  set <named variable> <value>          sets the named variable to value.

  show <token>
    breakpoints                         list breakpoints.
    coverage [true|false]               print each executed linenumber.
    env                                 show environment variables.
    version                             show NSA version.

nsa>
```

As you can see, most commands can take an abbreviated form e.g. *run* or just ‘r’. Code coverage can be achieved by issuing ‘*show coverage true*’. This will print each executed line, during the various flow-control commands like *run, next* or *step.* The *print* command will print named variables for the current scope unless specified with ‘*’, which includes all temporary variables (the ones that have not been declared with *local*). Tables will only be ‘dumped’ when specified directly, e.g. *print* mytable .
Please note that LUA code executed by the *thread* function will not be available for debug, since it actually runs in a separate LUA context.
You can set breakpoints by using a line-number or a function-name. If the function is in a different lua file (script/module) then specify the breakpoint with a *file-name:function-name* or *file-name:linenumber.*

Use arrow-up key to access the command history.