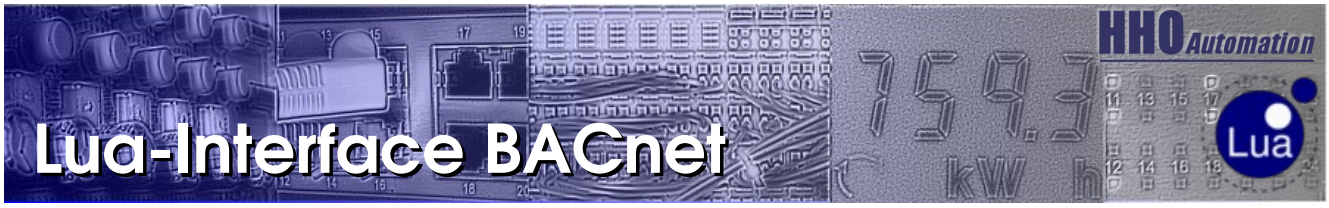

BACnet Lua-Interface alle BACnet Produkte

Lua ist eine eingebettete Script-Sprache und wird u.a in Industrieapplikationen genutzt. Sie ist einfach, effizient, erweiter- und portierbar.
Innerhalb der BACnet-Produkte ermöglicht das Lua-Interface dem Benutzer direkte Lese- und Schreibzugriffe auf alle Devices im BACnet-Netzwerk. Durch Syntaxerweiterungen sind Abhängigkeiten und Verknüpfungen unterschiedlicher BACnet-Objekte und deren Properties frei programmierbar und steigern deutlich die System-Funktionalität. Die Lua-Programme werden dabei in mehreren Tasks parallel ausgeführt. Mögliche Bespiele für Lua-Anwendungen:

- ✔ Lesen/Verknüpfen/Schreiben innerhalb/zwischen Devices ohne GLT/OWS
- ✔ Macroobjektierung (1 BACnet-Object <-> x Feld-IOs, > 5 BACnet-Properties)
- ✔ COV/Event-Notification Catching + Weiterverarbeitung
- ✔ freigestaltbare Benutzer-Interfaces Seriell/Netzwerk
- ✔ 'triviale' Kopplungen (Seriell/Netzwerk)
- ✔ freiprogrammierbare Datensammlungen (Dateisystem lokal/netzwerkweit)
- ✔ Benutzer-Benachrichtigungen (SMS, EMailing)
- ✔ vieles, woran wir heute noch nicht denken ;-)

Die nachfolgende Dokumentation erläutert die zusätzlichen Lua-Funktionen in unseren BACnet-Produkten wie Gateway, Controller und Watcher. Für eine detaillierte Beschreibung der Lua Scriptsprache und deren Syntax verweisen wir auf die LUA Homepage http://www.lua.org.

Lua-Interface für BACnet-Gateway und -Controller

Die Konfiguration erfolgt ausschließlich mit BACnet-Mechanismen und -Diensteverwaltung. Die Lua-Tasks werden als File-Objekte geladen und über Programm-Objekte gesteuert.
Jeweils einer Lua-Programmdatei ist ein Programmobjekt im EDE-File zugeordnet. Damit der Controller erkennt, das es sich um ein Lua-Programm handelt, sind die Instanznummern 1001 bis 1016 für diese Objekte reserviert. Beim ersten Einlesen des EDE-Files werden die zugehörigen Fileobjekte mit den Instanznummern 1001..1016 automatisch erstellt (siehe auch Beschreibung Konfiguration BACnet).

Das Programm selbst wird mit einem beliebigen Text-Editor geschrieben (Lua Syntaxunterstützung z.B. Notepad++/Win oder Geany für Win/Lin/OSX); anschließend wird die Datei über BACnet in das Fileobjekt geschrieben. Das zugehörige Programmobjekt steuert dann die Lua-Task (Run/Stop). Je nach Typ des Produktes können bis zu 8 (Bacnet-Watcher) bzw. 16 Lua-Programme parallel ausgeführt werden.

Lua-Interface für BACnet-Watcher

Auf dem BACnet-Watcher können bis zu 8 Lua-Tasks gleichzeitig aktiv sein. Die zugeordneten Dateien werden einfach über den HTML-Browser durch Dateiupload übertragen; die Namenssyntax sind wie folgt fest vorgegeben: **task01.lua** für Programm Nr.1 bis **task08.lua** für Programm Nr. 8.
Die Steuerung (Start/Stop) und Zustandsanzeige der Tasks erfolgt über eine eigene HTML-Seite, dabei wird die zweite Zeile in einem Lua-Programm als Kommentar für die Anzeigeinformation verwendet, Beispiel:

```
-- LUA-Task 01
-- Diese Zeile wird im BACnet-Watcher als Info angezeigt
...
```

Gerätespezifische Lua-Funktionen

LuaRun()

Jede Lua-Task wird von einem Programmobjekt gesteuert, über das Property ProgramChange kann die Task angehalten werden. Damit die Task die Anforderung erkennt, muß LuaRun() aufgerufen werden.
Rückgabewert *double* 1=ProgramRequestRun, 0=<>ProgramRequestStop.

```
while(LuaRun() == 1) do
  -- do something useful
end
```

LuaDelay(ms)

Läßt die Task für die Zeit in ms ruhen und gibt dem System Prozessorzeit. Der Wert muss mindestens 5 sein. LuaDelay prüft das zugehörige ProgramObject Property ProgramChange. Es wird empfohlen bei zyklischen Tasks immer die Funktion LuaDelay(ms) aufzurufen.
Rückgabewert *double* 1=ProgramRequestRun, 0=ProgramRequestStop.

```
if (LuaDelay(250) == 0) then
  return
else
  -- do something useful
end
```

LuaVarWrite(index, wert)

Schreibt den Doublewert *wert* in die globale Variable mit dem index zwischen 1 und 512. Jede Lua-Task hat Zugriff auf diesen Speicherbereich.
Rückgabewert: *double* Fehlercode 0=kein Fehler, -1=Fehler.

```
-- write the value 12.34 into Index 20
error = LuaVarWrite(20, 12.34)
if (error == 0) then
  -- do something useful
end
```

LuaVarRead(index)

Liest eine globale Variable mit dem Index zwischen 1 und 512. Jede Lua-Task hat Zugriff auf diesen Speicherbereich.
2 Rückgabewerte: *double* Fehlercode 0=kein Fehler, -1=Fehler, *double* Variablenwert.

```
 -- read the value 12.34 from Index 20
error, dvar = LuaVarRead(20)
if (error == 0) then
  -- do something useful with dvar
end
```

LuaTaskActive(index)

Prüft ob eine Task im Controller aktiv ist. Der Index ist die betreffende Tasknummer und muss einen Wert zwischen 1 und 8 bzw. 16 haben.
Rückgabewert *double* 1=Task ist aktiv (RunningState), 0=Task inaktiv oder nicht vorhanden.

```
if (LuaTaskActive(2) == 1) then
  -- Task 2 is running
else
  -- Task not available
end
```

LuaBACnPropWrite(device_id, obj_type, instance, property, prio, value, arrayindex)

Schreibt eine Objekteigenschaft mit vorgegebener Priorität. Bei device_id = -1 wird das Objekt im Gerät selbst (=localDevice) beschrieben. Der Wert value kann vom Typ „double“ oder „string“ sein. Die Priorität muss den Wert zwischen 1 und 16 haben. Ohne Angabe von arrayindex wird default '-1' gesetzt. Folgende BACnet Datentypen können geschrieben werden: Integer, Unsigned, Real, Double, Enumerated, Char_String, Octet_String und Bit_String.
Rückgabewert: *double* Fehlercode 0=Schreiben erfolgreich, -1=Fehler.

```
-- set PresentValue 15 in device 230
err = LuaBACnPropWrite(230, 2, 15, 85, 8, 15)
-- set LimitEnable to high and low
err = LuaBACnPropWrite(230, 2, 15, 52, 8, “11“)
-- reset PresentValue to default
err = LuaBACnPropWrite(230, 2, 15, 85, 8, “Null“)
```

LuaBACnPropRead(device_id, obj_type, instance, property, arrayindex)

Liest eine Objekteigenschaft. Bei device_id = -1 wird das Objekt im lokalen Device gelesen. Ohne Angabe von arrayindex wird default '-1' gesetzt. Es können alle Properties gelesen werden.
2 Rückgabewerte: *double* Fehlercode 0=kein Fehler, -1=Fehler (siehe auch LuaBACnGetLastError), *string* PropertyValue.

```
-- read property presentValue from device 230
err, sval =  LuaBACnPropRead(230, 2, 15, 85)
-- read property objectName from local device
err, sval = LuaBACnPropRead(-1, 2, 15, 77)
-- read objectcount from device 201, objectList[0]
err, sval = LuaBACnPropRead(201, 8, 201, 76, 0)
```

LuaBACnPropWriteIntern(obj_type, instance, property, value) *[*1]*

Schreibt eine Objekteigenschaft im lokalen Device über internen Zugang (Priorität 0). Der Wert value kann vom Typ „double“ oder „string“ sein. Folgende BACnet Datentypen können geschrieben werden: Integer, Unsigned, Real, Double, Enumerated, Char_String, Octet_String und Bit_String. Es können nur Properties beschrieben werden, die keine konfigurierte Feldreferenz besitzen.
Rückgabewert: *double* Fehlercode 0=Schreiben erfolgreich, -1=Fehler.

```
-- set PresentValue 15 of analog value 501 in local device intern
err = LuaBACnPropWriteIntern(2, 501, 85, 15)
-- set LimitEnable to high and low
err = LuaBACnPropWriteIntern(2, 501, 52, “11“)
```

LuaBACnPropReadIntern(obj_type, instance, property) *[*1]*

Liest eine Objekteigenschaft im lokalen Device über internen Zugang; es können alle Properties gelesen werden. 2 Rückgabewerte: *double* Fehlercode 0=kein Fehler, -1= Fehler (siehe auch LuaBACnGetLastError), *string* PropertyValue.

```
-- read property presentValue from local device intern
err, sval = LuaBACnPropReadIntern(2, 501, 85)
-- read property objectName from local device
err, sval = LuaBACnPropReadIntern(2, 501, 77)
```

LuaBACnGetLastError()

Gib den letzten Fehlercode nach einem Read- oder Write-Service zurück.
2 Rückgabewerte: *double* Errorclass, *double* Errorcode (siehe BACnet Spezifikation)

```
-- read a present value from Device 230
err = LuaBACnPropRead(230, 2, 15, 85)
if (err == -1) then
  errClass, errCode = LuaBACnGetLastError()
end
```

LuaFieldVarWrite(index, value) *[*1]*

Schreibt einen Wert vom Typ Double in eine konfigurierte Feldvariable (siehe Konfiguration EDE-Objects) mit der Nummer index; dabei wird der „Rohwert“ ins Feld geschrieben; evtl. Konvertierungen wie bei BACnet-Objekten gibt es hier nicht bzw. sind in Lua selbst zu erledigen.
Rückgabewert: *double* Fehlercode 0=Schreiben erfolgreich, -1=Fehler.

```
-- write value 1 to FieldVar 99
err = LuaFieldVarWrite(99,1)
-- write value 23.5 to FieldVar 2
err = LuaFieldVarWrite(2, 23.5)
```

LuaFieldVarRead(index) *[*1]*

Liest den Wert einer konfigurierten Feldvariablen (siehe Konfiguration EDE-Objects) mit der Nummer index; dabei wird der „Rohwert“ aus dem Feld gelesen; evtl. Konvertierungen wie bei BACnet-Objekten gibt es hier nicht bzw. sind in Lua selbst zu erledigen.
2 Rückgabewerte: *double* Fehlercode 0=kein Fehler, -1: Fehler, *double* Variablenwert.

```
-- read value from FieldVar 99
err, dval =  LuaFieldVarRead(99)
-- read value from FieldVar 2
err, dval = LuaFieldVarRead(2)
```

LuaDebugTask(p1, p2, p3... pn)

Schreibt die Parameter p1 bis pn als Text in die System DebugPipe /tmp/dbg. Ist man im System eingeloggt, kann man die Ausgabe über 'cat /tmp/dbg' auf das Terminal umleiten.

```
ErrClass,ErrCode = LuaBACnGetLastError()
LuaDebugTask(“BACnet Errorclass:“,errClass,“BACnet errorCode“,ErrClass)
```

LuaLogWrite(p1, p2, p3... pn)

Schreibt die Parameter p1 bis pn als Text in die Datei log_messages. Über den integrierten WebServer können die Einträge ausgelesen werden.

```
errClass, errCode =  LuaBACnGetLastError()
--  BACnet ErrorCode 30 means timeout
if (errCode  == 30) then
  LuaLogWrite(“Timeout, object [230,2,15] not available:“)
end
```

LuaTaskPrompt(p1, p2, p3... pn) *[*2]*

Schreibt die Parameter p1 bis pn als Text in die Taskübersicht, Spalte Description, der Lua Task Page. Hiermit können dann z.B. Fortschrittmarker oder Info-/Rückmeldungen zur Laufzeit der einzelnen Lua Programme angezeigt werden.

```
-- clean Event-Buffer
LuaEventBufferClear()
LuaTaskPrompt(" Eventbuffer:", "buffer cleared, ready for Events ..")
```

LuaCOVBufferRead()

Liest die COV-Notification Propertyeinträge aus einem Rundpuffer, die max. Puffergröße ist über bacnet.ini einstellbar (nur Gateway, im Watcher fix).
3 Rückgabewerte: *double* Fehlercode 0=kein COV, 1=COV vorhanden, -1=wie 1, aber Pufferüberlauf festgestellt, *string* DeviceObjectPropertyReference als komma-separiertem String im Format „deviceID,objectTyp,objectID,propertyID“, *string* Propertywert.

```
LuaCOVBufferClear()   -- init/reset Buffer

-- Loop 10ms
while (LuaDelay(10) == 1) do
  -- read buffer until empty
  repeat
    err, DevObjProp, PValue = LuaCOVBufferRead()
    if (err ~= 0) then
      if (err == -1) then
        LuaLogWrite("COV Buffer Overflow detected");
      end
      LuaLogWrite("COV: DevObjProp =", DevObjProp, ", Value =“, PValue)
    end
  until (err == 0)
end
```

LuaCOVBufferClear()

Initialisiert/löscht den COV-Notification Buffer; diese Funktion sollte vor Beginn von zyklische Ausleseroutinen für einen definierten Start verwendet werden.

LuaEventBufferRead()

Liest die empfangenen EventNotification Sevices aus einem Rundpuffer, die max. Puffergröße ist über bacnet.ini einstellbar (nur Gateway, im Watcher fix).

2 Rückgabewerte: *double* Fehlercode 0=kein Event, 1=Event vorhanden, -1=wie 1, aber Pufferüberlauf festgestellt, *string* BACnetEventInformation als Semikolon separiertem String im Format „ProcessIdentifier;InitiatingDeviceIdentifier;EventObjectIdentifier;TimeStamp;NotificationClass;Priority; EventType;“MessageText“;NotifiyType;AckRequired;FromState;ToState“ (siehe 135-2008 13.9.1)

```
LuaEventBufferClear() -- init/reset Buffer

-- Loop 10ms
while (LuaDelay(10) == 1) do
 -- read buffer until empty
 repeat
  berr, bevent = LuaEventBufferRead()
  if (berr ~= 0) then
     -- buffer overflow
     if (berr == -1) then
        LuaLogWrite(Prompt, "Error Event-Buffer overflow detected!")
     end
     -- capture event elements from string
     pid, devid, objtyp, objid, timest, nc, prio, etype, mtext,
     ntype, ack, froms, tos =
     string.match(bevent, "(%d+);(%d+);(%d+);(%d+);(.*);(%d+);(%d+);
                           (%d+);\"(.*)\";(%d+);(%d+);(%d+);(%d+);.*")
  end
 until (berr == 0)
end
```

LuaEventBufferClear()

Initialisiert/löscht den EventNotification Buffer; diese Funktion sollte vor Beginn von zyklische Ausleseroutinen für einen definierten Start verwendet werden.

LuaReadFavoriteList() *[*2]*

Liest die Objektverweise der aktuellen Favoritenliste, siehe Button/Seite Favorites.
2 Rückgabewerte: *double* Anzahl 0=keine Favoriten geladen, >0=Objektanzahl,
string Favorites als Semikolon separierte Objektliste im Format
„deviceId1,objectTyp1,objectId1;deviceId2,objectTyp2,objectId2;...deviceIdn,objectTypn,objectIdn;“.

```
-- get favorites list
count, favorites = LuaReadFavoriteList()
if count > 0 then
  text = count .. " Objects found ="
  for devId, objTyp, objId in string.gmatch(favorites,
                                  "(%d+),(%d+),(%d+);") do
    text = text .. string.format(" (%s|%s|%s)", devId, objTyp, objId)
  end
  LuaTaskPrompt(text)
else LuaTaskPrompt("no Favorites found") end
LuaDelay(5000) -- display 5s
```

LuaSingleTimer(timeout, func())

Startet einen Timer mit timeout in Sekunden. Ist die Zeit timeout abgelaufen, wird die Funktion func() aufgerufen.

HINWEIS: *Es steht für alle LuaTasks nur ein Timer mit dieser speziellen Funktion zur Verfügung.*
Wird in der Funktion func() der Timer nochmals aufgerufen, wird die Funktion func() zyklisch aufgerufen.

```
timeout = 5
-- this function will fire every 5 seconds
function MyTimer()
  LuaDebugTask("Timer event", os.date"%T")
  LuaSingleTimer(timeout)
end
-- initialize the timer
LuaSingleTimer(timeout, MyTimer)

while (LuaDelay(100) == 1) do
  -- do something useful in this loop
end
```

LuaSerPortOpen(port, mode, baud, charsize, stopbit, parity) *[*3]*

Öffnet eine serielle Schnittstelle im RAW-Mode: port 1-4, mode 0=RS232, 1=RS485,
Baudrate baud 300..115200, charsize 7 oder 8 Bit, stopbit 1 oder 2, parity 0=none, 1=even, 2=odd.
1 Rückgabewert: *double* Fehlercode 0=Öffnen erfolgreich, -1=falsche Parameterzahl / ungültige Portnummer, -2=Fehler beim Öffnen der Schnittstelle, -3=Schnittstelle belegt.

HINWEIS: *Die Schnittstelle muß hardwareseitig mittels Jumper entsprechend dem Modus konfiguriert sein (siehe Inbetriebnahme BACnGTW-SMARTi).*

```
-- open port 2: mode RS485 with 9600bd, 8Bit, 1 Stopbit, no Parity
err = LuaSerPortOpen(2, 1, 9600, 8, 1, 0)
if (err < 0) then
    -- handle error
else
    --port successful opened
end
```

LuaSerPortRead(port) *[*3]*

Liest von einer geöffneten Schnittstelle port 1-4 die empfangen Daten.
2 Rückgabewerte: *double* Wert >= 0 Anzahl der Empfangenen Zeichen, -1=falsche Parameter / Portnummer, -2=Fehler beim Lesen der Schnittstelle, -3=Schnittstelle von anderer Task belegt. s*tring* empfangene Daten (max. 4096 Bytes/Aufruf)

```
-- get data from port 2
count, data = LuaSerPortRead(2)
if (count < 0) then
    -- handle error
else if (count > 0) then
    -- count data received, compute data
else
    -- nothing received
end
```

LuaSerPortWrite(port, data, len) *[*3]*

Sendet eine Zeichenkette data mit der Länge len an eine geöffnete Schnittstelle port 1-4.
1 Rückgabewert: *double* Fehlercode 0=Senden OK, -1= falsche Parameter / ungültige Portnummer, -2=Fehler schreiben Schnittelstelle, -3=Schnittstelle belegt.

```
buffer = "some data to send"
-- send data to port 2
err = LuaSerPortWrite(2, buffer, 17)
if (err < 0) then
      -- handle error
end
```

LuaSerPortClose(port) *[*3]*

Schliesst die serielle Schnittstelle port 1-4.
1 Rückgabewert: *double* Fehlercode = 0 schließen OK, -1= falsche Paramter / ungültige Portnummer, -3 = Schnittstelle belegt.

HINWEIS: *Vor Beendigung der LUA-Task muss die Schnittstelle korrekt geschlossen werden damit diese wieder freigegeben wird, ansonsten bleibt diese Schnittstelle bis zum Neustart des Gateways belegt bzw. gesperrt.*

```
-- closing port 2
err  = LuaSerPortClose(2)
if (err < 0) then
      -- handle error
end
```

Abschließend ein Beispiel für eine zyklische Lua-Task 1:

```
-- Name task01.lua
-- Info: wenn Task 2 aktiv, lese global Var1 + schreibe remote PV

function PrintBACnetError(op, dev, objtype, instance, property)
 errClass, errCode = LuaBACnGetLastError()
 -- print out to logfile
 LuaLogWrite(op, "ErrorClass", errClass, "ErrorCode", errCode,
             dev, objtype, instance, property)
-- This will produce following output
-- date time Stack: LuaLog[01] Write ErrorClass x ErrorCode y 230 1 1 85 end

while (LuaRun() == 1) do
 -- check if the local Task 2 is active
 if (LuaTaskActive(2) == 1) then
        -- read out value if Task 2 is running
        err, readvalue = LuaVarRead(1)
        -- write a remote device object PresentValue
        if (err == 0) then
               err = LuaBACnPropWrite(230, 1, 1, 85, 9, readvalue)
               -- if there is an error then print it
               if (err == -1) then
                      PrintBACnetError("Write", 230, 1, 1, 85)
               end
        end
 end
 -- wait a little
 LuaDelay(500)
end
```

[*1] nicht für BACnet-Watcher
[*2] nicht für BACnet-Gateway
[*3] nur für BACnGTW-SMARTi