Ohjelmakirjastot

Yleistä

Yleistä

Sovellusohjelmilla on käytössään kaikki Lua-kielen standardi-kirjastot. Enimmäkseen ne ovat hyvin käyttökelpoisia, mutta suoritettaessa käyttöjärjestelmän shell komentoja esim. os.execute() kutsulla, tai massamuistia käytettäessä täytyy pitää mielessä, että käyttöjärjestelmä ei takaa komennolle laisinkaan suoritusaikaa, sopivassa tilanteessa jossa kaksi prosessia koettaa kirjoittaa samaaan tiedostoon, saadaan aikaa kilpailutilanne (race condition), joka aiheuttaa kyseisten prosessien totaalisen jumittumisen. Lisäksi tiedostoon kirjoittaminen ja lukeminen ovat usein prosesseja joiden nopeus voi vaihdella järjestelmän kuormitustilanteen ja muistipuskureiden täyttöasteen mukana jopa kymmenkertaisesti suorituskertojen välillä.

Lua ja erityisesti luaJIT mahdollistavat ainakin periaatteessa myös minkä vain käyttöjärjestelmästä löytyvän ominaisuuden tai jaetun kirjaston käyttämisen. Ne eivät kuitenkaan kuulu tämän dokumentin aihepiiriin.

Actiweb

SLC engineen on lisätty myös joitakin Plc ohjelmointia tukevia ohjelmakirjastoja, joita ei ole yleisesti jaossa. Nämä kirjastot on ladattu aina automaattisesti, eivätkä ne tarvitse *require* -kutsua.

SLC

sisältää SLC enginen tehtävien hallitsemiseen tarvittavia funktioita. Sovellusohjelmien tarvitsee käyttää näitä vain harvoin.

Tärkeä huomio!
Kutsut jotka vaikuttavat plc-prosessin suoritusparametreihin tai hakevat tietoa niistä eivät tavallisesti toimi laisinkaan mikäli niitä kutsutaan Luan **coroutine** säikeen sisältä, koska tällainen suoritussäie näyttää Slc-kirjaston kutsujen mielestä toiselta lua virtuaalikoneelta (prosessi toisen prosessin sisällä). Näin ollen Slc-kirjaston kutsu ei tiedä mihin plc-prosessiin kutsu kohdistuu.

Slc.enableWatchdog ()
Slc.disableWatchdog ()
Ei argumentteja

Koko järjestelmän watchdog ominaisuuden ottaminen käyttöön, tai asettaminen pois päältä. Huomaa, että monilla laitealustoilla ei ole mahdollista kytkeä watchdog ajastinta pois päältä sen käynnistämisen jälkeen. Kun watchdog ajastin on käynnistetty Slc.enableWatchdog() kutsulla, slc engine huolehtii automaattisesti ajastimen ajoittaisesta resetoinnista (kutsutaan usein vahtikoiran syöttämiseksi) niin kauan on käynnissä, eikä sovellusohjelmanm tarvitse välittää asiasta. Oikea tapa käyttää näitä funktiota on esimerkiksi kutsua toista niistä yhden kerran järjestelmän käynnistyksen yhteydessä.

Paluuarvot
true jos onnistui, *nil* jos kohdattiin virhe.

Esimerkki:
```
-- Set watchdog ON
   Slc.enableWatchdog ()
```

Slc.setWatchdogDelay (ms)
Slc.getWatchdogDelay ()
ms *Taskin vahtikoira-ajastimen viive millisekunteina*

Slc.error (strLevel, strMessage)

strLevel Virheen vakavuus: ”exception”, ”notify”, ”fault”, ”error”, ”critical”, ”fatal”

strMessage Virheen kuvausteksti

Ilmoittaa virhetilanteesta käyttäjälle, ja mikäli virhe on riittävän vakava, voi aiheuttaa myös PLC ohjelman siirtymisen virhe tilaan.

”error” taso aiheuttaa PLC ohjelman siirtymisen ”running” tilasta ”error” tilaan.

”critical” tai ”fatal” taso aiheuttaa PLC ohjelman suorituksen päättymisen, mikä yleensä aiheuttaa watchdog:n vuoksi koko laitteen uudelleen käynnistymisen.

Paluuarvot:
true mikäli onnistui, *nil* mikäli kohdattiin virhe (esimerkiksi virheellinen argumentti).

Esimerkki:
Slc.error(”error”, ”Houston, we had a problem”)

Slc.version ()
ei argumentteja

Lukee ja palauttaa järjestelmän ohjelmistoversion ja muita tietoja taulukkona. Taulukossa on kentät ”application” joka kertoo sovellusohjelman version (mm. monet plc kirjastot riippuvat tästä), ”kernel” joka kertoo käyttöjärjestelmän ytimen version ja ”plc” joka kertoo slc enginen binääritiedoston version.

Paluuarvot
taulukko jossa ohjelmistoversioiden tiedot.

Esimerkki:
print (”System kernel version: ” .. Slc.version().kernel)

Slc.createTask (strFile, strName, strMainCall, strSchMode, intDelay)

strFile lähdekooditiedoston koko polku

strName PLC tehtävän nimi

strMainCall tehtävän pääfunktion nimi

strSchMode Ajoitustila ”periocid” tai ”cyclic”

intDelay Ajoitusviive millisekunteina

Luo uuden PLC tehtävän annettujen parametrien perusteella. Tehtävä luodaan niin, että ensiksi tarkistetaan ovatko argumentit oikein, ja sen jälkeen käynnistetään uusi säie, ja viimeisessä vaiheessa ladataan ja käännetään annettu lähdekooditiedosto.

Paluuarvot:

true jos tehtävän luominen onnistuu, muuten *nil*.

Esimerkki:
Slc.createTask(”/opt/slc/prg/main.lua”, ”myTask”, ”main”, ”cyclic”, 500)

Slc.getSchedule ()
Slc.setSchedule (strMode)
strMode Haluttu ajoitustila, ”cyclic” tai ”periodic”

Funktioiden avulla voidaan lukea PLC ohjelman nykyinen ajoitustila, tai asettaa se halutuksi.

Slc.getSchedule palauttaa nykyisen tilan merkkijonona (”cyclic” tai ”periodic”).

Slc.setSchedule kutsulla se voidaan asettaa halutuksi,ja paluuarvo kertoo onnistuiko operaatio (true tai nil).

Esimerkki:

```
if Slc.getSchedule() ~= cyclic then
   Slc.setSchedule (”cyclic”)
end
```

Slc.getTiming ()
Slc.setTiming (intDelay)
intDelay Haluttu ajoitusviive

Funktioilla saadaan luettua PLC ohjelman nykyinen ajoitusviive, tai asetettua se halutuksi.

Slc.getTiming lukee viiveen ja palauttaa sen millisekunteina.

Slc.setTiming taas asettaa ohjelman viiveeksi halutun millisekuntimäärän. Palauttaa onnistuessa true, muutoin nil.

Slc.getTaskInfo ()
Slc.setName (strName)
strName Haluttu uusi PLC tehtävän nimi

Funktiolla getTaskInfo saadaan luettua PLC ohjelman tietoja, ja kutsu palauttaa taulukon jossa on kentät: ”name”, ”sourcefile”, ”callcounter”, ”runtime”, ”maincall”.

Name kenttä sisältää tehtävän nimen, sourcefile taas lähdekooditiedoston polun.

Callcounter kenttä on kokonaisluku joka kasvaa yhdellä joka kerran kun tehtävän pääfunktiota kutsutaan, eli se kertoo PLC ohjelman suorituskerrat.

Runtime kertoo kuinka kauan PLC ohjelman suorittaminen kestää, yksikkönä nanosekunti (PLC ohjelmien sisäinen ajoitus tehdään nanosekunteina).

SetName -kutsulla on mahdollista vaihtaa PLC ohjelman nimeä. Kutsu palauttaa true jos onnistuu, muutoin nil.

Esimerkki:
local info = Slc.getTaskInfo()
print (”task. ”..info.name..” running time: ”..info.runtime)

Slc.echo (strTxt)
strTxt *Tulostettava teksti*

Tulostaa tekstiä tai ohjelman tilaan liittyvää tietoa joka voidaan näyttää esimerkiksi grafiikka sivulla (mm. taskList -komponentti).

Paluuarvot
true mikäli onnistuu, muuten *nil*.

Esimerkki:
Slc.echo(n..” error on this run cycle”)

Slc.runRemote (strHost, strCmd)

strHost kohde laitteen IP-osoite tai DNS nimi.

strCmd Kohde koneessa suoritettava lua koodi

Tämän funktion avulla voidaan suorittaa lua komentoja toisessa SLC laitteessa (tcp portti 30001) ja lukea suoritetun lua-koodin paluuarvo takaisin ohjelmaan.

Funktiota voidaan käyttää hyvin monipuolisesti esimerkiksi tiedonsiirtomenetelmänä.

Huomioi! Tämä protokolla ei sisällä laisinkaan tietosuojaominaisuuksia! Jos tätä tiedonsiirtoprotokollaa käytetään julkisessa verkossa, se tulee suojata esimerkiksi SSH tunnelin avulla.

Esimerkki:
-- Lukee ulkolämpötilan LJH vakista
local te00 = Slc.runRemote (”192.168.0.201”, ”return Data.get(’ioPoints/TE00.pv’)”)

-- käynnistä poistopuhallin IV konehuoneessa

```
if not Slc.runRemote (”192.168.0.205”, ”return Data.set(’ioPoints/PK01/PF01.pv’, 1)”) then
    Slc.error (”fault”, ”Etäohjaus ei onnistu”)
```

```
end
```

System

Sisältää käyttöjärjestelmään ja laitteistoon liittyviä funktioita.

System.base64encode (strTxt)
System.base64decode (strTxt)
strTxt *Tulostettava teksti*

Koodaa ja dekoodaa tekstiä tai binääridataa base64 -muotoon. Kyseistä koodaustapaa käytetään usein binäärimuotoisen datan siirtämiseksi vain tekstimuotoa tukevan linjan ylitse – usein ascii 7-bit.

Paluuarvot
Palauttaa koodatun tai dekoodatun datan.

Esimerkki:
-- Lue binääri dataa tiedostosta ja koodaa se
local bin = file:read (”*a”)
local coded = System.base64encode(bin)

System.shell (strCmd)
strCmd *Suoritettava komento*

Suorittaa komennon järjestelmän shellissä (synkronisesti, eli kutsuva käyttöjärjestemän prosessi odottaa että komento on suoritettu kokonaan) ja palauttaa komennon stdout:iin tulostaman tekstin riveittäin taulukossa. Palautettu taulukko on numeraalisesti indeksoitu 1 eteenpäin, ja yksi tulosteen rivi vastaa yhtä taulukon riviä.

Palauttaa *nil* jos komentoa ei voitu suorittaa, ja tyhjän taulukon jos komento ei tulosta mitään std out:iin.

Komento palauttaa toisena paluuarvona suoritetun komennon paluuarvon, eli n.s. exit code:n.

Esimerkki:
-- hakemiston tiedostot
-- huom! että kaksi ensimmmäistä aina . Ja ..
-- ne pitää ohittaa

```
local dir, ec = System.shell (”ls /tmp/*”)
if #dir > 2 and ec == ”0” then
    for i = 3, #dir do
        print (dir[i])
```

```
    end
end
```

System.log (prio, strCmd)

prio Loki merkinnän kiireellisyys. Kokonaisluku välillä 0 .. 7 jossa 0 vähiten kiireellinen (debug).
strCmd Tulostettava teksti

Kirjoittaa merkinnän järjestelmä lokiin.

Paluuarvot
true mikäli onnistuu, muuten *nil*.

Esimerkki:
Slc.log(0, ”This is my system log info”)

System.nanosleep (intDly)
intDly Viive nanosekunteina

Vastaa POSIX kutsua nanosleep(). Asettaa kutsuvan säikeen uneen (minimissään) annetuksi ajaksi.

Esimerkki:
-- Plc ohjelma odottaa 1 millisekunnin
Slc.nanosleep (1000000)

System.nanotimer ()
Ei argumentteja

Palauttaa järjestelmän tarkkuusajastimen nykyisen arvon. Ajastin on laskuri (uint64), joka alkaa juosta 0:sta kun käyttöjärjestelmä käynnistyy. Sen laskee nanosekuntteja, ja sen todellinen tarkkuus on usein noin 40 nanosekunnin luokkaa (ARM cortex A8).

Esimerkki:
print (”Current hires timer value: ”.. System.nanotimer ())

System.parseUrl (strUrl)
strUrl Url teksti

Parsii URL merkkijonon, ja palauttaa URL eri kentät talukossa.

Esimerkkinä:
URL: https://areena.yle.fi/tv/ohjelmat/sarjat?t=uusimmat

Parsittu taulukko:
URL_taulukko = {
   protocol = ”https”,
   user = ””,
   pass = ””,
   address=”areena.yle.fi”,
   port=””,
   fullpath=”/tv/ohjelmat/sarjat”,
   path={”tv”, ”ohjelmat”, ”sarjat”},
   paramstring=”t=uusimmat”,
   params={t=”uusimmat”}
}

Toinen esimerkki:

URL:
http://name:secret@www.google.com:8080/this/is/path/file.xml?param1=1&meaning=42

Parsittu taulukko:
{
   protocol = "http",
        user = "name",
         pass = "secret",
         address = "www.google.com",
         port = "8080",
         fullpath = "/this/is/path/file.xml",
         path = {"this", "is", "path", "file.xml" },
         paramstring = "param1=1&meaning=42",
         params = {
            param1 = "1",
            meaning = "42"
         }
}

Virhetilanteessa funktio voi palauttaa arvon *nil*.

Esimerkki:

```
local parsed = System.parseUrl ( urlString )
if parsed.protocol == ”http” then
   -- Handle http request
end
```

System.encodeUrl (strData)

System.decodeUrl (strData)
strData *Tulostettava teksti*

Kun käsitellään URL tekstejä, on itse lokaattorin lopussa annettavien parametrien arvot koodattava niin sanotulla prosentti koodauksella. Näillä funktioilla voidaan tehdä parametrien arvojen prosenttikoodaus, ja dekoodaus.

Paluuarvot
koodattu tai purettu teksti.

Esimerkki:
-- Palauttaa meik%C3%A4%0A
local encoded = System.encodeUrl (”meikä”)

System.serialize (data)
data *Serialisoitava data (merkkijono, luku, taulukko)*

Tekee lua objekteille niin sanotus sarjoituksen, eli serialisoinnin.

Täma muunnos tarkoittaa että objekti muutetaan takaisin lähdekoodi -muotoon. Tätä muunnosta tarvitaan esimerkiksi silloin kun lua taulukko halutaan siirtää tcp yhteyden ylitse toiseen laitteeseen, tai vain IPC-kanavan lävitse laitteen muistissa toiseen prosessiin, tai tallentaa tiedostoon.

Serialisoitu objekti on helppoa palauttaa takaisin ohjelmassa käsiteltäväksi, koska se voidaan ”ajaa” normaalissa lua tulkissa.

Vastaa suurinpiirtein javascriptin JSON.stringify() kutsua.

Tämä funktio on käyttökelpoinen paitsi tiedonsiirrossa ja tallentamisessa, myös ohjelmien debuggauksessa, koska miltei mikä tahansa objekti voidaan tulostaa tekstimuodossa vaikkapa konsoliin ohjelmoijan tarkasteltavaksi.

Paluuarvot
Annettu objekti tekstimuodossaan merkkijonona, tai nil mikäli kutsuttiin virheellisillä arvoilla.

System.importCSV (strCsv [, strDlm])

strCsv Tab eroteltu data
strDlm Sarakeet erottava merkki, oletus on \t eli tabulaattori

Tämä funktio luo CSV muotoisesta – tai oletusarvoisesti TAB-erotellusta tekstistä – taulukon ja palauttaa sen.

Nimestään huolimatta oletus sarake-erottimelle on ’\t’.

Luotavan rivin nimen oletetaan löytyvän sarakkeesta, jonka nimi on ”dataname”, ”pointname”, ”rowname”, ”datapoint” tai ”keyname” – kaikki edellämainitut ovat synonyymejä.

Esimerkki:
```
csv = [[name    description       pv
data1       tunnit       7.0
data2       minuutit       15.0
data3       sekunnit       23.0
data4       paivat       1.0
]]

local d = System.importCSV (csv)

>> taulukon d sisältö:
d = {
   data1 = {description = ”tunnit”, pv = 7},
   data2 = {description = ”minuutit”, pv = 15},
   data3 = {description = ”sekunnit”, pv = 23},
   data4 = {description = ”paivat”, pv = 1}
}
```

System.archInfo ()

Funktio palauttaa taulukossa laitteiston käyttöjärjestelmän ja prosessoriarkkitehtuurin.

Palauttaa taulukon, jossa rivit
- os Käyttöjärjestelmän nimi. Joitakin tavallisia arvoja ovat mm. **Ubuntu, Builtroot, Debian.**
- arch Prosessoriarkkitehtuuri. Tyypillisiä arvoja ovat **armv7l** (beagle bone) ja **x86_64**

Esimerkki:
```
local d = System.archInfo (csv)
print (”Käyttöjärjestelmä: ”.. d.os)
```

System.getNetworkInterfaces ()

Funktio palauttaa taulukossa järjestelmän verkkosovittimet ja niiden asetukset.

System.pid ()

Funktio palauttaa kutsuvan prosessin (yleensä slc engine) process ID (eli pid) numeron. Tämä on se tekninen tunnus, jolla käyttöjärjestelmä tunnistaa prosessit, ja jonka avulla voi lähettää mm. signaaleita prosessien välillä.

System.sendSig (pid, signal)
pid (int) prosessin ID numero
signal (int) signaali joka lähetetään (kts. posix singals)

Funktion avulla voi lähettää signaalin käyttöjärjestelmäprosessien välillä. Palauttaa *true* onnistuessaan, ja *nil* mikäli kutsussa ilmeni virhe.

BitOps

Sisältää perus binäärioperaatioita, ja muistipuskureihin liittyviä funktioita. Jotkin näistä funktioista ovat päällekkäisiä luaJIT:n **bit** -kirjaston kanssa, eikä niiden välillä ole suurta toiminnallista eroa.

Huomaa, että monista operaatioista on olemassa eri versioista (esim. not16 ja not32) joiden erona on, että operaatioiden aikana lukuja käsitellään funktion nimessä mainitulla tarkkuudella – binääri-operaatioiden tapauksessa muunnos tehdään aina unsigned muotoon (ei etumerkkiä).

BitOps.not16 (v)
BitOps.not32 (v)
v Arvo jolle halutaan tehdä binäärinen NOT operaatio.

Suorittaa annetun luvun biteille loogisen NOT operaation, eli invertoi ne.

Esimerkki:
-- Tulos on 0xFF
local r = BitOps.not16 (0xFF00)

BitOps.and16 (v1, v2)
BitOps.and32 (v1, v2)
v1 Binäärisen JA -operaation syöteluku.
v2 Binäärisen JA -operaation syöteluku.

Suorittaa annettujen lukujen välillä AND operaation.

Esimerkki:
-- Tulos on 0xFF
local r = BitOps.and16 (0xFFFF, 0xFF)

BitOps.or16 (v1, v2)
BitOps.or32 (v1, v2)
v1 Binäärisen OR operaation syöte.
v2 Binäärisen OR operaation syöte.

Suorittaa annettujen lukujen välillä OR operaation.

Esimerkki:
-- Tulos on 0xFFFF
local r = BitOps.and16 (0xFF00, 0xFF)

BitOps.xor16 (v1, v2)
BitOps.xor32 (v1, v2)
v1 Binäärisen XOR operaation syöte.
v2 Binäärisen XOR operaation syöte.

Suorittaa annettujen lukujen välillä n.s. poissulkevan OR operaation (eng. exclusive or), jota usein merkitään XOR lyhenteellä.

Esimerkki:
-- Tulos on 0xFF
local r = BitOps.and16 (0xFFFF, 0xFF00)

BitOps.shr16 (v, n)
BitOps.shr32 (v, n)
BitOps.shl16 (v, n)
BitOps.shl32 (v, n)
v Siirto operaation kohdeluku.
n Siirrettävien bittien määrä.

Siirtää luvun v bittejä n kappaletta joko oikealle (shr) tai vasemmalle (shl). Vasemmalle siirto kasvattaa luvun arvoa, ja oikealle siirto pienentää sitä.

Esimerkki:
-- Siirtää bittejä puolikkaan tavun verran,
-- joka vastaa yhtä heksadesimaali numeroa
-- Tulos on 0xFF00
local r = BitOps.shl16 (0xFF0, 4)

BitOps.createBuffer (n)
n Luotavan puskurin koko tavuina.

Luo uuden raakadatapuskurin.

Esimerkki:
-- Luo puskurin jonka koko on 32 tavua.
local buf = BitOps.createBuffer (32)

BitOps.fillBuffer (b, v, i, n)
b Puskuri jolle operaatio tehdään (luotu createBuffer kutsulla)
v Arvo jota puskuriin kirjoitetaan (huom! Arvo muutetaan BYTE tyyppiseksi, lukualue 0x0 .. 0xFF)
i Indeksi josta täyttö aloitetaan (oltava >= 0)
n Kirjoitettavien tavujen määrä (puskurin indeksistä i lähtien).

Kutsu kirjoittaa BitOps.createBuffer -kutsulla luotuun muistipurkuriin tiettyä argumenttina annettua tavua, ja sitä voidaan käyttää esimerkiksi nollaamaan puskurin muistipaikat.

Huomaa että indeksoint seuraa lua käytäntöä, eli puskurin ensimmäinen tavu on indeksissä 1!

Esimerkki:
-- Luo puskurin ja nollaa sen kaikki tavut
local buf = BitOps.createBuffer (32)
BitOps.fillBuffer (buf, 0x0, 0, 32)

BitOps.deleteBuffer (b)
b Puskuri jolle operaatio tehdään (luotu createBuffer kutsulla)

Tuhoaa muistipuskurin ja vapauttaa sille varatun muistin.

Esimerkki:
-- Luo puskurin ja sitten tuhoaa sen

```
local buf = BitOps.createBuffer (2048)
if buf then
    BitOps.deleteBuffer (buf)
end
```

BitOps.getBufferLen (b)
b Puskuri jonka koko halutaan tietää

Palauttaa argumenttina annetun muistipuskurin pituuden – eli sille varatun muistin määrän tavuina.

Esimerkki:
-- Luo puskurin ja näyttä sen koon

```
local buf = BitOps.createBuffer (2048)
if buf then
    print (”Buffer is ” .. BitOps.getBufferLen(buf).. ” bytes long”)
end
```

BitOps.setBufferLen (b)
b Puskuri jonka pituus halutaan asettaa

Muuttaa jo luodun muistipuskuri kokoa. Operaatio tehdään varaamalla puskurille uusi muistialue ja kopioimalla vanha sisältö uuteen – jos uusi puskuri on pienempi kuin vanha, kopioidaan vain se osa joka uuteen puskuriin mahtuu.

Esimerkki:
```
-- Luo puskurin ja muutaa sen koon pienemmäksi
local buf = BitOps.createBuffer (2048)
BitOps.setBufferLen (buf, 32)
```

BitOps.getByte (b, i)
BitOps.setByte (b, i, v)

BitOps.getWord (b, i)
BitOps.setWord (b, i, v)

BitOps.getDWord (b, i)
BitOps.setDWord (b, i, v)

BitOps.getString (b, i)
BitOps.setString (b, i, v)

BitOps.getFloat32 (b, i)
BitOps.setFloat32 (b, i, v)
b Puskuri jolle operaatio tehdään
i Luettavan tai arvon alkukohdan indeksi (tavuja)
v Kirjoitettava arvo

Näillä funktioilla voidaan kirjoittaa ja lukea muistipuskurista yksittäisiä tavuja, sanoja, kaksoisanoja, merkkijonoja tai IEEE koodattuja liukulukuja.

Huomaa että indeksoint seuraa lua käytäntöä, eli puskurin ensimmäinen tavu on indeksissä 1!

Esimerkki:
```
-- Luo puskuri
local buf = BitOps.createBuffer (2048)
BitOps.setString(buf, 1, ”Handling buffer”)
local byte = BitOps.getByte(buf, 1)        -- Lukee merkin ’H’ ascii koodin 0x48
```

BitOps.asBitArray (v, [n])
v Luku joka halutaan muuttaa

n Vastauksen sisältämä TAVU määrä (ei pakollinen)

Muuttaa annetun luvun lua taulukoksi, joka sisältää riveillä annetun luvun bitit.

Argumentilla n voidaan määrätä montako tavua vastauksessa on (1 tavu vastaa 8 bittiä, eli 8 riviä).

Esimerkki:
-- Luo puskurin ja muutaa sen koon pienemmäksi
local arr = BitOps.asBitArray(0xF0, 1)

>> tulos:
arr = {1, 1, 1, 1, 0, 0, 0, 0}

BitOps.arrayAsDWord (t)
t Luvuksi muunnettava taulukko

Käänteinen operaatio BitOps.asBitArray () -kutsulle. Muuntaa annetun bitti taulukon takaisin luvuksi.

Esimerkki:
-- v saa arvon 0xF0
local v = BitOps.arrayAsDWord ({1, 1, 1, 1, 0, 0, 0, 0})

BitOps.fillArray (v, n, [i])
v Arvo jota taulukon soluihin kirjoitetaan
n Kuinka monta kertaa arvo kirjoitetaan taulukkoon
i Indeksi josta kirjoittaminen aloitetaan (ei pakollinen)

Luo uuden lua -taulukon ja täyttää sen annetulla luvulla.

Esimerkki:
-- luo taulukon jossa 512 riviä arvoina 1
local t = BitOps.fillArray(1, 512)

BitOps.convertTo (v, t)
v Arvo joka muunnetaan
t Muoto merkkijonona, johon luku halutaan muuttaa ”int8”, ”uint8”, ”int16”, ”uint16”, ”int32”, ”uint32”, ”int64”, ”uint64”, ”float32”.

Muuttaa luvun annettuun datatyyppiin. Tätä muunnosta tarvitaan melko harvoin, mutta joskus on välttämätöntä tulkita luku esimerkiksi uint16 tyypissä.

Esimerkki:
local v = 0xFFFF

```
local int16 = BitOps.convertTo (v, ”int16”)    -- Tulos -32767
local uint16 = BitOps.convertTo (v, ”uint16”)    -- Tulos 65535
local overFlow = BitOps.convertTo ( (v+1), ”uint16”)    -- Tulo nolla
```

BitOps.binaryDWordAsFloat (dw)
BitOps.binaryFloatAsDWord (f)
dw   Kaksoissana joka halutaan muuttaa float32 koodatuksi.
f   Float32 luku josta halutaan saada raaka binääriesitys.

Näillä kutsuilla on mahdollista muuttaa esimerkiksi tiedostosta tai sarjaliikenneportista luettu binääriesitys takaisin liukuluvuksi, ja toisinpäin.

Esimerkki:

```
-- puskuriin luetussa datassa on liukuluku,
-- palautetaan se binääri esityksestä käytettävään muotoon
b = BitOps.getDWord (buf, 5)
f = BitOps.binaryDWordAsFloat (b)
```

XML

Sisältää XML datan käsittelyyn tarvittavia funktioita. Perustuu TinyXML2 -kirjastoon.

Xml.parseAsLuaDOM (strXml)
Xml.loadAsLuaDOM (strFile)
strXml Merkkijono joka sisältää XML dataa
strFile Tiedosto josta XML data ladataan

Parsii XML annetun datan, ja palauttaa sen lua taulukkona.

Tuloksena syntyvä lua taulukko seuraa seuraavaa DOM -mallia mukailevaa rakennetta:

-- Esimerkki XML data:

```
xmlData = [[
<kirjat>
   <kirja muoto=”kovakantinen”>
      <kirjailija>Pekka Meikäläinen</kirjailija>
      <vuosi>1975</vuosi>
   </kirja>
   <kirja muoto=”pokkari”>
      <kirjailija>Matti Meikäläinen</kirjailija>
      <vuosi>1974</vuosi>
   </kirja>

</kirjat> ]]
```

-- Funktiokutsu:
local d = Xml.parseAsLuaDOM (xmlString)

>> tulos

```
d = {
   _name = ”kirjat”,
   _attr = {},
   _text = ””,
   kirja = {
      _name = ”kirja”,
```

```
    _attr = {muoto=”pokkari”},
    _text = ””,
    kirjailija={
        _name = ”kirjailija”,
        _attr = {},
        _text = ”Matti Meikäläinen”
    },
    vuosi={
        _name = ”vuosi”,
        _attr = {},
        _text = ”1974”
    }
    [1] = {
        _name = ”kirjailija”,
        _attr = {},
        _text = ”Matti Meikäläinen”
    }
    [2] = {
        _name = ”vuosi”,
        _attr = {},
        _text = ”1974”
    }
},
[1] = {
    _name = ”kirja”,
    _attr = {muoto=”kovakantinen”},
    _text = ””,
    kirjailija={
        _name = ”kirjailija”,
        _attr = {},
        _text = ”Pekka Meikäläinen”
    },
    vuosi={
        _name = ”vuosi”,
        _attr = {},
        _text = ”1975”
    },
    [1] = {
        _name = ”kirjailija”,
        _attr = {},
```

```
        _text = ” Pekka Meikäläinen”
      },
      [2] = {
        _name = ”vuosi”,
        _attr = {},
        _text = ”1975”
      }
    },
    [2] = {
      _name = ”kirja”,
      _attr = {muoto=”pokkari”},
      _text = ””,
      kirjailija={
        _name = ”kirjailija”,
        _attr = {},
        _text = ”Matti Meikäläinen”
      },
      vuosi={
        _name = ”vuosi”,
        _attr = {},
        _text = ”1974”
      },
      [1] = {
        _name = ”kirjailija”,
        _attr = {},
        _text = ”Matti Meikäläinen”
      },
      [2] = {
        _name = ”vuosi”,
        _attr = {},
        _text = ”1974”
      }
    }
}
```

Yllä olevasta esimerkistä käy ilmi, että tuloksena yksinkertaisesta XML datasta on melko monimutkainen lua taulukko. Syitä siihen on monia; XML dokumentin perusyksikkö on n.s. node, eli solmu. XML dokumentti koostuu määritelmän mukaan yhdestä juuri elementistä – esimerkissä nimeltään kirjat – joka voi puumaisesti sisältää minkä tahansa määrän alisolmuja. Koska nämä

solmut voivat olla nimeltään identtisiä, täytyy XML dokumenttia parsiessa säilyttää paitsi solmun nimi, myös niiden järjestys. Koska lua taulukoiden tapauskessa on usein näppärämpää viitata taulukon alkioihin nimellä kuin indeksillä, tässä tapauksessa molemmat.; Lapsi solmut lisätään lua-taulukkoon paitsi juoksevasti numeroiden, myös solmu nimellään, jolloin viimeinen lisätty solmu jää voimaan. Esimerkin tapauksessa ”kirjat.kirja” osoituksen takaa löytyy Matti Meikäläisen kirjoittama kirja.

Tässä mallissa säilytetään siis kaikki XML dokumenttiin sisältyvä tieto.

Xml.loadAsTable (strXml)
Xml.parseAsTable (strFile)
strXml Merkkijono joka sisältää XML dataa
strFile Tiedosto josta XML data ladataan

Lataa annetun XML datan ja palauttaa tuloksen yksinkertaisena taulukkona. Tuloksena syntyvä lua -taulukko on yksinkertaisempi kuin DOM mallia seuraava, mutta se ei esimerkiksi ymmärrä solmujen attribuutteja, tai saman nimisiä lapsisolmuja.

Esimerkki:
-- Esimerkki XML data:

```
xmlData = [[
<kirjat>
   <kirja muoto=”kovakantinen”>
      <kirjailija>Pekka Meikäläinen</kirjailija>
      <vuosi>1975</vuosi>
   </kirja>
   <kirja muoto=”pokkari”>
      <kirjailija>Matti Meikäläinen</kirjailija>
      <vuosi>1974</vuosi>
   </kirja>

</kirjat> ]]

-- Funktiokutsu:
local d = Xml.parseAsLuaDOM (xmlString)

>> tulos
d = {
   kirjat={
      kirja={
         kirjailija=”Matti Meikäläinen”,
```

```
        vuosi=”1974”
      }
    }
}
```

Xml.serialize (o, r)
o Lua objekti joka muutetaan XML muotoon.
r Dokumentin juurisolmun nimi

Muuttaa hyvin suoraviiveisesti lua-taulukon XML muotoon.

Huomaa, että lua-taulukon numeraaliset indeksit eivät käänny XML muotoon oikein. Funktio muuttaa ne <1>, <2>, .. jne nimisiksi solmuiksi, mutta nämä eivät ole sallittua XML:ää.

Esimerkki:

-- Esimerkki XML data:

```
data = {
   description = ”testi data”,
   pv = 90.5,
   polarity = ”normal”
}

local xml = Xml.serialize (data, ”testNode”)

>> tulos
xml = [[
<testNode>
   <description>testi data</description>
   <pv>90.5</pv>
   <polarity>normal</polarity>
</testNode>
]]
```

sqlite

Toteuttaa rajapinnan sqlite3 -tietokantojen käsittelyyn sovellusohjelmista.

Sqlite.connect (strDB)
strDB Tietokanta tiedosto joka halutaan avata

Avaa tietokantatiedoston käsittelyä varten

-- Avaa tietokannan
local bd = Sqlite.connect (”/opt/slc/data/myData.sql”)

Sqlite.disconnect (hDB)
hDB Avatun tietokannan kahva

Sulkee yhteyden tietokantatiedostoon

-- Avaa ja sulkee tietokannan
local bd = Sqlite.connect (”/opt/slc/data/myData.sql”)
Sqlite.disconnect (db)

Sqlite.query (hDB, strQuery)
hDB Avatun tietokannan kahva
strQuery SQL query joka halutaan suorittaa

Ajaa SQL queryn tietokannassa ja palauttaa tuloksen taulukkona (tai nil mikäli kyselyssä tapahtuu virhe).

Jos kysely ei palauta dataa, palautetaan tyhjä taulukko.

Huomaa! Kaikki SQL queryt täytyy päättää ; -merkkiin, muuten ne palauttavat virheen.

-- Suorita yksinkertainen kysely tietokantaan
local bd = Sqlite.connect (”/opt/slc/data/myData.sql”)
local r = Sqlite.query(db, ”select * from myTable;”)
Sqlite.disconnect (db)