

# Actiweb

# sovellusohjelmointi

- Esitietoa
- Prosessit ja tehtävät
- libalarms ja alarmServer
- Sovellusohjelmat
- libhvac ja hvacServer
- Ohjelmointityökalut
- libhvacex
- Pistetietokanta
- OPC-UA
- Ohjelmakirjastot
  - Yleistä
  - SLC
  - System
  - BitOps
  - XML
  - sqlite
- alarm aux

# Esitietoa

Actiweb järjestelmään on luotu valmiiksi monia erisovelluksiin tarkoitettuja ohjelmakirjastoja, jotka eivät ole sisäänrakennettuja slcengine binäärissä, vaan ovat lua-kielisinä laitteen ../lib/ ja autorun hakemistoissa. Joskus ne pitää ladata lua-ohjelmaa tehtäessä **require** käskyllä. Näitä kirjastoja kutsutaan Actiweb järjestelmässä sovelluskirjastoiksi.

# Prosessit ja tehtävät

Ohjelmoitavien logiikoiden tapaan myös Actiweb järjestelmän tärkeimpiä osiaon sovellusohjelmien ajaminen säännöllisesti ja luotettavasti. Jokaista tällaista itsenäistä ohjelmaa kutsutaan ohjausjärjestelmien yhteydessä usein tehtäväksi, mutta käyttöjärjestelmien tapauksessa samantapaista ohjelmistokokonaisuutta kutsutaan joskus prosessiksi

Actiweb **ei** seuraa IEC61131 standardia ohjausjärjestelmien ohjelmoinnista, vaan sovellusohjelmia kirjoitetaan Lua kielellä. Toisaalta, Lua kieli muistuttaa hieman edellämäintun standardin sisältämää ST-kieliltä. Muilta osin Actiweb-järjestelmä muistuttaa paljon IEC-standardin kuvaamaa ohjausjärjestelmää, jossa sovellusohjelmat käynnistetään toimimaan n.s. taskeina, joita suoritetaan tarkasti määrättyllä suoritusvälillä niin kauan kuin järjestelmä on toimintakykyinen.

Actiweb tukee nykyisessä versiossa kahta eri sovellusohjelman ajoitusmallia: "cyclic" ja "periodic", joissa on lopulta vain pieni ero.

**Periodic** tarkoittaa tasavälistä ajoitusta, siinä tehtävä käynnistetään aina samalla aikavälillä (mikäli prosessorin suorituskyky sen sallii). Eli mikäli tehtävän aikaväliksi määritetty 1000 ms (1 sekunti), ja tehtävän suorittaminen kestää 250 ms, odottaa kyseinen tehtävä 750 ms lepotilassa ennen seuraavaa käynnistystä. Tämä "lepoaika" lasketaan jokaisen suorituskerran jälkeen, ja järjestelmä koettaa pitää suorituskertojen (sykliä) alkamishetket mahdollisimman tarkasti asetetun mukaisina.

**Cyclic** tarkoittaa että tehtävän suorittamisen jälkeen tehtävä asetetaan lepotilaan annetuksi ajaksi - esimerkiksi mainittu 1000 ms. Tällä tavalla voidaan varmistaa että prosessorille jää aikaa suorittaa muita toimia, ja sitä voidaan hyvin käyttää mikäli tehtävän tarkka ajoitus ei ole aivan kriittinen.

Useinkaan suoritustilan valinnalla ei ole kovin suurta merkitystä, mikäli itse ohjelmat on kirjoitettu niin että ne eivät oleta täysin tasaista suoritusväliä. Usein tärkeintä on, että ohjelma ajetaan riittävän usein ohjattavaan prosessiin nähden. Sovellusohjelmia kirjoitettaessa kannattaa huomata myös se, että esimerkiksi määrätyn pituisen viiveen tekeminen ei vaadi tarkkaa ohjelman suoritusväliä, vain riittävän sovelluksen kannalta riittävän tiheän suoritusvälin. Prosessori ja käyttöjärjestelmä pitävät taustalla käynnissä hyvin käyttökelpoista nanosekunti-tason tarkkuusajastinta ja reaaliaikakelloa, joiden avulla esimerkiksi ajastukset ja viiveet on helppo tehdä. Niitä voi hyödyntää sovellusohjelmissa API kutsuilla:

```
System.nanotimer ()  
os.time()  
os.date()
```

Jos tarkkaa ohjelmien ajoitusta tarvitaan, **periodic** -tilassa ohjelman ajoituksen huojunta (niin sanottu jitter) on Cortex A8 prosessorille käännetyllä versiolla suuruusluokassa +/- < 50 us

(mikrosekuntia) mikäli prosessorin kuormitus pysyy alle 80 %. Pitkän ajan vakaus riippuu täysin kellosignaalin lähteen vakaudesta (esimerkiksi AM3358 prosessorin tapauksessa sisäisen oskillaattorin stabiilius +/- 50 ppm).

Tavallinen tapa tehdä uusi ohjelmaprosessi, on luoda lua -päätteinen tiedosto /opt/slc/prg/run - hakemistoon. Käynnistyksen yhteydessä järjestelmä skannaa edellämainitun hakemiston lua-päätteiset tiedostot, ja käynnistää ne oletusasetuksilla, jotka ovat:

**cyclic** -suoritustapa, 1000ms suoritussväli.

Näitä asetuksia on mahdollista muuttaa käyttämällä API kutsuja:

**Slc.setSchedule (strMode)**

**Slc.setTiming (iDelay)**

Joskus on hyödyllistä käynnistää yhdestä ohjelmatiedostosta useampia ohjelmaprosesseja – joita kutsutaan usein PLC järjestelmien yhteydessä tehtäviksi, tai englanninkielisellä termillä "task". Uuden ohjelmaprosessin saa käynnistettyä käyttämällä API kutsua:

**Slc.createTask ( strFile, strName, strCall, strMode, delay)**

Jokainen task eli tehtävä on siis oma käyttöjärjestelmä prosessinsa, tai tarkemmin sanottuna säie. Jokaiselle säikeelle on varattu oma muistialueensa, ja Lua ohjelmat toimivat täysin toisistaan riippumatta. Se tarkoittaa sitä, että säikeet eivät suoraan näe toistensa muuttujia tai funktioita. Tämä tekee tavallisesti ohjelmista vakaampia, ja helpottaa niiden kirjoittamista. Helpoin keino tietojen vaihtamiseen Lua ohjelmien välillä on pistetietokanta. Monimutkaisempi mutta mahdollinen tapa on käyttää väliaikaistiedostoa tai käyttöjärjestelmän "nimettyjä putkia", jotka matalalla käyttöjärjestelmätasolla ovat tavallaan myös väliaikaisia tiedostoja.

Samanaikaisesti käynnissä olevien Lua ohjelmien maksimimäärä on sama kuin käyttöjärjestelmän prosessien maksimimäärä, joka selviää tiedostosta

*/proc/sys/kernel/pid\_max*

Raja on usein 32 768 prosessia, mutta tyypillisesti muut suorituskykyyn liittyvät seikat, kuten muistin määrä tai prosessoriteho tulee vastaan ennen varsinaista prosessien määrää.

Käyttökelpoisia Linux komentoja:

**free** Ilmoittaa vapaan muistin määrän

**df** Ilmoittaa vapaan levytilan määrän

Sovelluksesta riippuen, sekä muistinkulutusta että vapaata levytilaa hyvä seurata, sillä mikäli sovellusohjelmisto kerää paljon esimerkiksi historia-dataa, voi levytila tai muisti loppua helposti kesken.

# libalarms ja alarmServer

Toteuttaa pistetyypit **alarm** ja **alarmGroup** joka on tarkoitettu hälytysten käsittelyyn.

Kirjasto tulee normaalisti valmiiksi asennettuna, ja toimii taustalla muista ohjelmista riippumatta.

Kirjasto käynnistää ohjelman alarmServer, joka tekee hälytyksiin liittyviä operaatioita, kuten hälytysviivettä ja laukaisee hälytystapahtumia.

Hälytyspisteet toimivat niin, että hälytyksen tila voidaan lukea **av** kentästä, jossa arvo 1 tarkoittaa pisteen olevan hälyttävässä tilassa, ja arvo nolla kertoo että hälytys on poistunut, eli se on pisteen normaali tila. Kun hälytyspisteen **pv** kenttä menee samaan arvoon kuin hälytyspisteen **alarmValue** kenttä, menee piste hälytystilaan **toAlarmDelay** kentässä määritetyn viiveen kuluttua (yksikkö on sekuntia). Kun sitten **pv** kentän arvo muuttuu jälleen erisuureksi **alarmValue** kentän kanssa, palautuu piste **toNormalDelay** viiveen jälkeen normaaliksi.

Jos pisteen kuittaustilaa halutaan seurata, se pidetään ylhäällä **alarmAcked** kentässä. Siinä tilat menevät niin, että 1 tarkoittaa hälytyksen olevan kuitattu, ja 0 tarkoittaa kuittaamatonta. Tämä kenttä muuttuu arvoon 0 joka kerran kun **av** kenttä saa arvon 1.

Hälytyspisteillä on myös **alarmStatus** kenttä, joka on aina samassa arvossa **av** kentän kanssa.

Kun **alarm** tyyppinen piste menee hälytystilaan, sen **alarmTime** ja **alarmDate** kentät asetetaan osoittamaan nykyistä aikaa, eli ne osoittavat edellisen ajanhetken kun jossa piste on käynyt hälytystilassa.

## Yhteistoiminta HVAC kirjaston kanssa

HVAC kirjasto laajentaa hälytyspisteiden toimintaa niin, että lisäämällä alarm -tyyppiseen pisteeseen sopivat kentät, hälytyspiste toimii automaattisesti mm. alaraja-, yläaraja-, säätövika-, tai ristiriitahälytyksenä.

## Jatkohälytykset

Mikäli hälytyksistä halutaan ilmoitusviesti esimerkiksi sähköpostin tai tekstiviestin avulla, osaa alarmServer hoitaa myös niiden lähettämisen.

Ilmoitusviestien vastaanottajat, ja vastaaottajien vaihtaminen esimerkiksi kellonajan tai viikonpäivän mukaan tehdään hälytys ryhmien avulla. Jokainen hälytys josta halutaan antaa ilmoitusviesti, tulee liittää sopivaan hälytysryhmään (**alarmGroup** pisteet).

Tämän kun hälytykset on liitetty ryhmiin, valitaan hälytysryhmästä mitkä tapahtumat aiheuttavat viestin lähtemisen. Tämä valitaan **enableAckEvent**, **enableToAlarmEvent** ja

**enableToNormalEvent.** Näiden selitteet ovat samassa järjestyksessä: lähetä viesti kun hälytys kuitataan, lähetä viesti kun hälytys menee päälle ja lähetä viesti kun hälytys poistuu.

Lisäksi viestiryhmälle voidaan ohjata eri tiloihin pv kentän avulla, ja antaa selitteet näille tiloille stateTexts kenttän taulukossa. Kun viestiä lähetetään, pv kentän arvon perusteella valitaan mitä vastaanottajalista käytetään, jolloin tila 0 voisi olla vaikkapa päivä, ja tila 1 voisi olla yö. Nyt vastaanottajalistaan (**recipientList**) voidaan luoda kaksi riviä, joista ensimmäinen lista vastaa yö tilaa, ja seuraava päivä tilaa.

Kun vastaanottajalista avataan ikkunaan klikkamalla **recipientList** kenttää hiirellä, voi uuden vastaanottajalistan tehdä painamalla uloimmalla tasolla '+' painiketta. Vastaanottajalista voidaan nyt nimetä **name** kentän avulla, ja lisätä uusia vastaanottajia listaan painamalla sisempää '+' painiketta. Myös vastaanottajat listan sisällä voidaan nimetä name kentässä. Osoite tai puhelinnumero johon viesti lähetetään annetaan **address** kentässä.

Teknisesti **address** kenttän arvon odotetaan aina olevan URL-muotoinen, mutta puhelinnumeron ja sähköpostin tapauksessa se ei ole välttämätöntä, eli järjestelmä tunnistaa normaalisti puhelinnumeron ja sähköpostiosoitteen, ja täydentää URL-osoitteen.

Mahdolliset lähetysmuodot (URL muotoilu):

**rut://[puhelinnumero]**

Lähetää tekstiviestin teltonika RUT modeemien kautta. Vaatii oikeat asetukset /sys/settings/smsOverHttp tietokantapisteeseen.

**relay://[ip osoite]**

Lähetää hälytysviestin toisen Actiweb CPU:n kautta.

**sms://[puhelinnumero]**

Lähetää viestin tekstiviestinä.

**email://[sposti@osoite.fi]**

Lähetää viestin sähköpostilla. Laitteen sähköpostiasetusten täytyy olla aseteltu.

Viestien muotoilua voidaan hallita valitsemalla sopiva viestipohja **alarmGroup** pisteen **subjectTxtFile** ja **bodyTxtFile** kenttien avulla.

Tiedosto jota viestipohjana käytetään on normaali tekstitiedosto, mutta viestin lähettämisen yhteydessä pohjaan täydennetään <TAG> -muotoisten tagien tilalle hälytyspisteen tai hälytysryhmän tietoja. Tarkemmin sanottuna, tageissa käytettävänä ovat kaikki viestin lähettämisen käynnistäneen hälytyspisteen kentät saman nimisinä, mutta suurilla kirjaimilla kirjoitettuna (esim. <ALARMDATE>, <DESCRIPTION>) sekä kaikki hälytykseen liittyvän hälytysryhmän kentät G\_ etuliitteen avulla (esim. <G\_DESCRIPTION>, <G\_PV>). Lisäksi käytettävissä on hälytyspisteen nimi <ID> tagin avulla.

# Sovellusohjelmat

Sovellusohjelma ja tehtävä ovat tässä dokumentissa oikeastaan synonyymejä. Käyttäjän kannalta sovellusohjelma sisältää lua koodin lisäksi myös mm. käyttöliittymän grafiikkasivut, joita ei käsitellä tässä dokumentissa.

## Tehtävien eli task:ien luominen

Mikäli käytetään vakio versiota käynnistys skriptistä, on tehtävien luominen järjestelmään hyvin helppoa, sillä käynnistys skripti etsii kaikki ".lua" -päätteiset tiedostot **/opt/slc/prg/run/** hakemistosta, ja käynnistää jokaisen niistä omana tehtävänä. Mikäli jossakin yhteydessä on tärkeitä hallita näiden ohjelmien käynnistysjärjestystä, se voidaan tehdä nimeämällä ohjelmatiedostot sopivasti. Järjestelmä nimittäin ensiksi hakee kaikkien hakemistossa sijaitsevien tiedostojen nimet, järjestää ne aakkosjärjestykseen – tai oikeastaan numerojärjestykseen ASCII merkitötaulukon mukaisesti pienimmästä suurimpaa. Näin ollen antamalla tiedostonimille alkuliitteet vaikkapa 01 .. 99 saadaan ohjelmien käynnistysjärjestys halutuksi. Esimerkin tapauksessa tiedosto joka alkaa 01 ladataan ja suoritetaan ensimmäisenä, ja tiedosto joka alkaa 99 ladataan viimeisenä.

Name	Size	Type	Date Modified
01-alarmServer.lua	317 bytes	Lua script	14.08.2019 at 14.31.44
01-dbServer.lua	8,6 KiB	Lua script	Yesterday at 12.06.49
01-device.lua	1,3 KiB	Lua script	14.08.2019 at 14.31.44
50-bacnetClient.lua	5,8 KiB	Lua script	14.08.2019 at 14.31.44
50-bacnetServer.lua	4,6 KiB	Lua script	14.08.2019 at 14.31.44
50-fileIO.lua	2,2 KiB	Lua script	14.08.2019 at 14.31.44
70-curlio.lua	4,9 KiB	Lua script	14.08.2019 at 14.31.44
70-fdxClient.lua	3,2 KiB	Lua script	14.08.2019 at 14.31.44
70-mbusMaster.lua	4,2 KiB	Lua script	14.08.2019 at 14.31.44
70-modbusInit.lua	4,0 KiB	Lua script	14.08.2019 at 14.31.44
70-modbusSlave.lua	5,1 KiB	Lua script	14.08.2019 at 14.31.44
70-reporting.lua	2,3 KiB	Lua script	14.08.2019 at 14.31.44
80-energy.lua	8,6 KiB	Lua script	14.08.2019 at 14.31.44
80-fdxServer.lua	2,2 KiB	Lua script	14.08.2019 at 14.31.44
90-trendlogs.lua	9,8 KiB	Lua script	14.08.2019 at 14.31.44

15 items (67,2 KiB), Free space: 2,9 GiB

Kuvassa on näkyvissä Actiweb laitteen /opt/slc/prg/run/ hakemistossa olevat automaattisesti käynnistyvät autorun ohjelmat.

Kuten sanottua, uusia task:eja eli tehtäviä on mahdollista luodaan lisäämällä uusi ".lua"-päätteinen tiedosto /opt/slc/prg/run/-hakemistoon. Kun tällainen automaattisesti käynnistettäv ohjelma ladataan, ja järjestelmä luo siitä taskin, annetaan sille aluksi seuraavat oletusasetukset:

- Suoritusväli 1000 ms
- Ajoitustapa cyclic (eli suorituskertojen välillä odotetaan 1000 ms)
- Nimi on tiedoston nimi

Näitä oletusarvoja on mahdollista muuttaa Slc kirjaston kutsuilla **Slc.setTaskName ()**, **Slc.setSchedule ()**, **Slc.setTiming ()**. Tätä hakemistoa kutsutaan lyhyesti autorun hakemistoksi.

Latausprosessi toimii siten, että ensiksi kääntäjä lataa ja kääntää autorun-hakemistossa olevan lähdekooditiedoston. Mikäli kyseinen tiedosto viittaa muihintiedostoihin esimerkiksi "require" käskyllä, ne ladataan ja käännetään samalla hetkellä kun kääntäjä törmää näihin viittauksiin. Alkuperäisen tiedoston kääntäminen jatkuu kun viitattu tiedosto on käsitelty.



Kun tiedosto on käännetty tavukoodiksi laitteen muistiin, se suoritetaan kokonaisuudessaan yhden kerran. Tämän jälkeen käynnistyy varsineinen ajoitettu suoritussykli, jossa actiweb ohjelmisto kutsuu asetetulla suorituvälillä ohjelman niin sanottua pääfunktiota. Autorun hakemistosta käynnistetyillä ohjelmilla se on aina nimeltään "main". Tämä nimitys mukailee monia ohjelmointikieliä kuten C, C++, Java tai C#.

Tämä suoritustapa tarkoittaa sitä, että pääfunktion ulkopuolinen ohjelmakoodi suoritetaan yhden kerran käännösvaiheen jälkeen, eli siellä voidaan hoitaa esimerkiksi erilaisia alustukseen liittyviä toimenpiteitä, kuten määrittää asetuksia tai globaaleita muuttujia.

Alla on esimerkki hyvin yksinkertaisesta Lua kielisestä sovellusohjelmasta, joka antaa hälytyksen mikäli mittaus ylittää raja-arvon. Ohjelma olettaa että pistetietokannassa on olemassa "TE20" ja "TE20\_HI\_AL" nimiset tietokantapisteen.

Esimerkki 1:

*/opt/slc/prg/run/example1.lua*

```
Slc.setSchedule ("periodic")
Slc.setTiming (1000)
Slc.setTaskName ("simpleAlarmTask")

function main()
  local m = Data.getReal ("TE20.pv")
  local hilimit = Data.getReal ("TE20.hi")
  local hyst = 0.5
  if m > hilimit then
    Data.set ("TE20_HI_AL.pv", 1)
  elseif m < (hilimit - hyst) then
    Data.set ("TE20_HI_AL.pv", 0)
  end
end
```

Yllä oleva ohjelma käynnistyy siis automaattisesti omaksi prosessikseen kun tiedosto on luotu, ja näkyy siten myös käyttöliittymässä system→setting -sivun alaosan tehtävälistassa. Tuossa listassa näytetään **Slc.setTaskName ()** kutsulla asetettu nimi.

On hyvä ymmärtää kuinka lua tiedostot ladataan ja suoritetaan. Koska lua on käyttäjän kannalta tulkattu kieli, näyttää tilanne siltä että sovellusohjelmat (ja muutkin komennot) suoritetaan suoraan tekstitiedostosta tai komentiriviltä, joka on täysin poikkeavata esimerkiksi C-kieleen verrattuna. Todellisuudessa lua-koodi käännetään samalla hetkellä kun käyttäjä pyytää koneelta että lähdekooditiedosto ajetaan. Tiedosto käydään lävitse, ja mikäli siinä ei ole syntaksi-virheitä, siitä tehdään RAM muistiin niin sanottu tavukoodi "möhkäle" - eli chunk. Tavallisesti tätä tavukoodikäännöstä ei kirjoiteta levyille. Tämän jälkeen perinteisen lua (tai python) ympäristön tapauksessa tulkki ryhtyy sitten suorittamaan tätä tavukoodia, ja osa virheistä paljastuu vasta

tässä vaiheessa. Slc engine käyttää kuitenkin lua ohjelmien suorittamiseen niin sanottua JIT kääntäjää, jolloin tavukoodi käännetäänkin suoraan natiiviksi konekieliseksi ohjelmakoodiksi, jonka prosessori sitten suorittaa. Teoriassa ohjelman kääntäminen kestää tässä tapauksessa hieman kauemmin, mutta toisaalta, ohjelmakoodin suorittaminen on hyvin nopeata.

## Funktiot ja globaalit muuttujat

Alla olevassa esimerkissä näytetään muutama hieman edistyneempi rakenne.

Esimerkki 2:

*/opt/slc/prg/run/example2.lua*

```
Slc.setSchedule ("periodic")
Slc.setTiming (1000)
Slc.setTaskName ("mySecondTask")

counter = 0

function inc(i)
    return i+1
end

function main()
    counter = inc(counter)
    Slc.echo ("Round ".. counter )
end
```

Ensimmäinen tärkeä seikka on counter muuttuja. Sen edessä ei ole local sanaa, jolloin se määrittyy globaaliksi muuttujaksi, ja on esittelyn jälkeen käytössä kaikkialla ohjelmassa. Globaaleilla muuttujilla on lisäksi tärkeä piirre, että ne **säilyttävät** arvonsa suorituskertojen välillä, eli niin kauan kuin Lua suoritussympäristöä ei alusteta uudelleen. Tämä tapahtuu pääasiassa silloin, kun koko Actiweb ohjelma uudelleenkäynnistetään – eli esimerkiksi painetaan restart -painiketta käyttöliittymässä, tai koko laitteisto uudelleenkäynnistyy. Joitakin laskureita ja viiveitä voi siis aivan hyvin tehdä jopa ylläolevalla tavalla.

Toinen huomionarvoinen piirre on funktio **inc()** – tämä liittyy itse lua-kieleen. Ohjelmien ylläpidon ja uudelleenkäytettävyyden kannalta olisi hyvä tapa, että samaa ohjelmakoodia kirjoiteta kuin yhden kerran. Tätä varten useimmissa ohjelmointikielissä on jokin tapa luoda aliohjelmia, jotka tekevät tietyn. Lua kielessä ne kuvataan function-avainsanan avulla, ja ohjelmointikielestä ja tilanteesta riippuen niitä voidaan kutsua fuktioiksi, rutiineiksi tai proseduureiksi. Kun ohjelman toistuvista osista tehdään funktio, eräs suurimmista hyödyistä on, että siinä olevaa virhettä ei tarvitse korjata kuin yhteen paikkaan. Toisaalta, kun rutiini nimetään järkevästi, se myös

yksinkertaistaa sitä ohjelman kohtaa, josta rutiinia kutsutaan. Se taas on ohjelman monimutkaisuuden hallinnan kannalta korvaamatonta, ja myös eräs proseduraalisen ohjelmointitavan keskeisistä ajatuksista.

## Oliot

Ohjelman ymmärrettävyyden kannalta on usein hyödyksi, kun sekä data, että toiminnallisuus (eli rutiinit) saadaan liimattua yhteen, jolloin tuloksena on olio.

Vaikka tämän oppaan tarkoituksena ei ole opettaa ohjelmointia yleisesti, eikä lua kieltä erityisesti, vain esittää mitä lua kielisten ohjelmien kirjoittaminen Slc engine ympäristöön vaatii, käymme seuraavassa lävitse olioiden (eng. objects) luomisen perusteet.

Lua kielessä oliot perustuvat prototyyppeihin ja **constructor** funktioihin (kuten myös esimerkiksi ECMA script -kielessä). C++ ja Java kielissä taas oliot luodaan n.s. luokkien pohjalta, mikä on hieman byrokraattisempi lähestymistapa, ja sopii paremmin vahvasti tyyppitettyihin kieliin.

Esimerkki 2:

*/opt/slc/prg/run/example3.lua*

```
Slc.setSchedule ("periodic")
Slc.setTiming (1000)
Slc.setTaskName ("mySecondTask")

-- Lets create object prototype cSimplePump
cSimplePump = {}
cSimplePump.new = function (s, idDI, idDO, idAL)
    local c = {}
    c.idDI = (idDI or "")
    c.idDO = (idDO or "")
    c.idAL = (idAL or "")
    c
    local c.state = 0 -- off by default

    c.setState = function (s, nState)
        if type(nState) ~= "number" then
            return false
        elseif nState < 0 then
            return false
        elseif nState > 1 then
            return false
        else
```

```

    s.state = nState
    return true
end
end

c.runLogic = function (s)
    -- Control conflict alarm
    local DI = Data.getReal (s.idDI)
    local DO = Data.getReal (s.idDO)
    if DI ~= DO then
        Data.set ( s.idAL, 1)
    else
        Data.set ( s.idAL, 0)
    end

    -- Control logic
    if s.state == 1 then
        Data.set ( s.idDO, 1)
    else
        Data.set ( s.idDO, 0)
    end
end

return c
end

-- Create pumps
PU01 = cSimplePump:new ("PU01_DI.pv", "PU01_DO.pv", "PU01_AL.pv")
PU02 = cSimplePump:new ("PU02_DI.pv", "PU02_DO.pv", "PU02_AL.pv")
PU03 = cSimplePump:new ("PU03_DI.pv", "PU03_DO.pv", "PU03_AL.pv")

function main()
    -- Handle all pumps
    if Data.getReal ("PU_ENABLED.pv") > 0 then
        PU01:setState (1)
        PU02:setState (1)
        PU03:setState (1)
    else
        PU01:setState (0)
        PU02:setState (0)
    end
end

```

```
    PU03:setState (0)
end

PU01:runLogic ()
PU02:runLogic ()
PU03:runLogic ()
end
```

Ylläoleva esimerkki on pyritty pitämään niin yksinkertaisena kuin mahdollista, ja sen on tarkoitus havainnollistaa sitä, että

- Miten objektityyppi luodaan - tässä tapauksessa cSimplePump konstruktori.
- Miten konstruktorin avulla luodaan uusia olioita. Tässä tapauksessa cSimplePump tyyppisiä olioita.
- Miten noita olioita on mahdollista käyttää.

ylläolevassa esimerkissä näytetään siis ensiksi, kuinka oliotyyppille varataan ensiksi tyhjä taulukko, ja sitten määritellään n.s. konstruktori, jonka avulla luokan oliot rakennetaan. Konstruktorin nimellä ei sinänsä ole mitään merkitystä, vaan tärkeitä on toiminnallisuus; Konstruktori on funktio, joka palauttaa aina kutsuttaessa tietyllä tavalla rakennetun olion. Konstruktori voi ollaa parametreina olion alustamiseen vaikuttavia tietoja, kuten tässä tapauksessa pistetunnuksia, jotka täytetään luotavaan olioon konstruktorissa.

Konstruktorin sisällä uutta oliota ryhdytään kasaamaan luomalla yhtä taulukko 'c'. Ensiksi siihen määritellään data-alkiot joita jokaisella tämän tyyppisellä oliolla on, ja sen jälkeen olion metodit, eli funktiot jotka muokkaavat olion omaa tilaa (eli dataa), ja joita kutsumalla oliota voidaan käyttää.

Olioiden kanssa puuhaillessa on hyvä muistaa niiden tärkeimmät edut; rajapinnat ja tiedon piilottaminen; Nämä seikat ovat tavallaan saman asian kaksi puolta. Plc ohjelmoinnissa olioita ovat muistuttaneet IEC61131 standardin kuvaamat Funktioblokit, mutta toisaalta, niistä puuttuu monia tärkeitä piirteitä, kuten juuri metodit. Vaikka lua ei kielenä sitä varsinaisesti rajoitakaan, ei olioiden sisäisiin muuttujiin (tässä tapauksessa eism. idDO tai status) saisi koskaan viitata suoraan, vaan niille tiedoille joita pitäisi päästä muuttamaan ulkoa päin, tulisi tehdä set() ja get() -metodit (eli funktiot). Näin voidaan varmistaa että olion tila (eli sen sisäiset muuttujat) pysyvät tietyissä rajoissa. Toisaalta, metodit tarjoavat selkeän rajapinnan olion käyttämiseen. Yllä olevassa esimerkissä tehtyä cSimplePump oliota voisi laajentaa esimerkiksi toteuttamaan vuorottelu automaattisesti, ja se olisi edelleen ohjelmoijalle yhtä helppo käyttää; kutsutaan olion run() metodia tasaisin väliajoin. Kaikki vuorottelun aiheuttama monimutkaisuus jää piiloon olion sisään.

## Lua on dynaaminen ohjelmointikieli

Heikosti tyypitetyt dynaamiset ohjelmointikielet – kuten Lua, Python ja Javascript – sisältävät piirteitä, jotka voivat vaatia vahvasti tyypitettyihin kieliin tottuneelta ohjelmoijalta hieman totuttelua ja ajattelutapojen muutosta.

Kenties suurin muutos on se, että koska muuttuja `myData` voi sisältää rivillä 5 merkkijonon, rivillä 20 taulukon, ja rivillä 50 numeraalisen arvon, kääntäjä ei ennen ohjelman suorittamista antaa virheilmoitusta mikäli muuttuja sisältää väärän tyyppistä tietoa, tai jos sitä ei ole olemassa ollenkaan. Tämä aiheuttaa ongelmia kun muuttujien arvoja vertaillaan toisiin arvoihin, vakioihin, tai niille koetetaan tehdä matemaattisia operaatioita. Muuttujan sisältämän datan tyyppin voi tarkistaa käyttämällä `type()` kutsua, ja se voi palauttaa arvot `"string"`, `"number"`, `"boolean"`, `"table"`, `"function"`, `"thread"`, `"userdata"` tai `"nil"`. Nämä ovat Lua kielen tuntemat datatyytit.

Ohjelmakoodissa voi olla rivi

```
myData = myData + 1
```

Ja mikäli `myData` sisältää tuossa vaiheessa arvon `false`, aiheutuu siitä virhe `"runtime error"`.

Dynaamisissa kielissä on tyypillinen piirre, että muuttujat ja objektit ovat olemassa vain niin kauan kuin niitä tarvitaan, eikä niitä tarvitse esitellä etukäteen millään tavalla. Muuttuja luodaan, kun sille annetaan arvo. Ja toisaalta, ja se lakkaa olemasta samalla hetkellä, kun sen arvoksi asetetaan *nil*. Kääntäen, muuttuja on olemassa, jos sen arvo on jotakin muuta kuin *nil*.

Esimerkki:

```
if myData ~= nil then
    print ("Variable exists")
else
    print ("variable does not exists")
end
```

Esimerkki:

```
if type (myData) == "string" then
    -- do something
end
```

Lua kielessä muuttujien joustavuutta kannattaa käyttää hyödyksi, ja haittoja voi minimoida käyttämällä loogisia operaatioita. Niiden avulla ohjelmien rakennetta saa usein yksinkertaistettua, eikä aina ole tarvetta käyttää if-käskyjä:

Esimerkki, jos *myData* on *nil* (ei olemassa) käytetään arvoa 0 sen tilalla

```
myData = (myData or 0) + 1
```

Esimerkki:

```
myData = (type(myData) ~= 'number') and 0 or myData
```

Yllä olevassa esimerkissä tarkistetaan onko myData:n tyyppi numero, ja jos ei, annetaan sille arvo 0, muutoin se pitää nykyisen arvonsa. Voisi sanoa, että yllä olevassa esimerkissä arvoa 0 käytetään muuttujan oletusarvona, jota käytetään mikäli muuttujaa ei ole olemassa, tai se on eri tyyppiä kuin pitäisi.

Lua sisältää myös *tonumber* ja *tostring* nimiset komennot jolla tulkki tai kääntäjä koettavat muuttaa muuttujan arvon joko numeroksi tai merkkijonoksi. Tämä muunnos tosin täytyy yleensä yhdistää ehdolliseen sijoitukseen, kutsut voivat palauttaa arvon nil mikäli muunnos ei onnistu - esimerkiksi jos koetetaan muuttaa taulukkoa numeroksi.

Not like this

```
myData = tonumber (myData)
```

but do it like this

```
myData = (tonumber (myData) or 0)
```

Monista kielistä löytyvää ehdollista sijoitusta (eng. *ternary operator*) ei Lua kielestä löydy. Sen sijaan, vastaavan toiminnallisuuden saavuttaa Lua kielessä usein käyttämällä *and* ja *or* operaatioita luovasti yllä olevaa esimerkkiä mukaillen.

C-kielessä käytetty ehdollinen sijoitus (ternary operator) näyttää tältä (**ei toimi Lua kielessä**):

```
int i = (r > 10) ? 1 : 0;    // i gets value 1 if r is grater than 10
```

Ja vaikka täysin vastaavaa rakennetta ei lua kielessä suoraan olekkaan, sitä vastaava toiminnallisuus ja hyvin saman kaltainen luettavuus on saavutettavissa *and* ja *or* operaatioilla. Tässä yhteydessä tulee kuitenkin kiinnittää erityistä huomiota suoritussyjestykseen, jotta operaatio toimii aina odotetulla tavalla.

```
local i = (r > 10) and 1 or 0    -- i gets value 1 if r is greater than 10
```

Samalla logiikalla voidaan tätä rakennetta käyttää myös seuraavilla tavoilla

```
local i = (r == nil) and 0 or i
```

```
local i = (type(r) == "string") and r or "none"
```

```
local i = (type(r) == "number") and r or -999
```

Tässä rakenteessa tärkeimmät mm. suoritusrakenteesta johtuvat piirteet ovat, että *and* operaatio palauttaa arvon, joka annetaan sen oikealla puolella (eli järjestyksessä jälkimmäisen), jos molemmat puolet ovat totuusarvoltaan *true*. Sellaisia arvoja ovat kaikki muut paitsi *false* ja *nil*. Or - operaatio palauttaa taas järjestyksessä ensimmäisen arvon, joka on totuusarvoltaan *true*.

Täten, yllä oleva rakenne palauttaa *and* operaation jälkimmäisen arvon, mikäli molemmat puolet ovat tosia, ja toisaalta, *or* operaation oikean puolen jos *and* operaatio palauttaa arvon *epätosi*.

Lisää tietoa tästä mm. lua-käyttäjien wiki sivulla:

<http://lua-users.org/wiki/TernaryOperator>



# libhvac ja hvacServer

Kirjasto sisältää kiinteistöautomaatiossa paljon käytettyjä toimintoja, kuten PID-säädin, viikkokello- ja kalenteriohjauksen, viiveajastimet ja säätökäyrät.

Kirjasto käynnistää hvacServer prosessin, joka suorittaa pistetietokannassa operaatiot kirjaston luomille pistetyypeille.

## hvacPIDController

Tämä pistetyyppi toimii PID säätimenä. Säätimeen voi määritellä haluamansa määrän säätöportaita, ja jokaiselle voi määrittää haluamansa suhdealueen, sekä lähdön minimin ja maksimin, sekä määrittää sen joko säätösuunnaltaan suoraksi tai käännetyksi (jolloin minimi ja maksimi toimivat nurinpäin).

Jos säätimessä on monia portaita käytössä, ja varsinkin tilanteessa jossa osa portaista on invertoitu, pitää ohjelmoijan muistaa asettaa säätimen **bias** kenttä sopivaan arvoon. Kyseinen kenttä määrittää pisteen, jossa säätimen portaiden lähdöt ovat käynnistyshetkellä. Haluttu **bias**-piste voidaan hakea laskemalla portaiden lähtöjen summa haluttuun käynnistyspisteeseen.

Jos säätimessä on kaksi porrasta, ja niiden lähdöt ovat alueella 0% - 100%. Säätimen ensimmäisen portaan saa puoliksi auki antamalla bias arvoksi 50, tai täysin auki antamalla bias-arvoksi 100. Jos myös seuraava porras halutaan vaikkapa 30 % auki, lasketaan pohjalle 1. portaan täysi säätövara (100%) ja sen jälkeen 2. portaan haluttu ohjaus eli 30 %. Tuloksena saadaan bias-arvo 130 (%).

Sama logiikka toimii useampienkin portaiden kanssa. Portaan vaikutus bias-arvoa laskettaessa on minimi-asennon ja maksimi-asennon erotus.

Koska säädin lähtee normaalisti käyntiin tilanteesta jossa kaikki portaavat ovat minimissä (pois lukien P-termin vaikutus), ja koska portaan **invertointi** kääntää portaan lähdön toiminnan, ovat invertoidut portaan tästä johtuen säätimen käynnistyessä usein lähellä maksimiarvoa. Tämä on useimmiten ei toivottu ilmiö, ja se saadaan korjattua laskemalla sopiva bias-arvo, jonka avulla säädin käynnistyykin esimerkiksi ensimmäisen säätöportaan jälkeen, jolloin bias arvoksi tulee yksinkertaisesti ensimmäisen säätöportaan lähdön maksimi - tarkemmin sanottuna ensimmäisen portaan maksimin ja minimin erotus.

Pisteen kentät:

**bias** Säätimen toimintapiste käynnityksessä.

**const\_I** Integrointiaika sekunteina (viritysparametri).

**const\_D** Derivoinnin vahvistus (viritysparametri)

<b>deadZone</b>	Integroinnin kuollut alue.
<b>input</b>	Säätimen takaisinkytkentätieto, eli mittausarvo.
<b>inputId</b>	Tietokantapiste, josta säätimen mittaustieto luetaan. Haluttaessa voidaan jättää tyhjäksi, ja kirjoittaa input kenttä vaikkapa ohjelmassa.
<b>enabled</b>	Säätimen käyntilupa. Arvossa 1 säädin algorimia suoritetaan, ja arvossa 0 säätimen lähdöt asetetaan default arvoihin.
<b>enabledId</b>	Tietokantapiste josta säätimen käyntilupa haetaan. Voidaan jättää tyhjäksi, ja kirjoittaa enabled kenttää esimerkiksi ohjelmassa.
<b>setpoint</b>	Säätimen asetusarvo, johon input arvoa koetetaan säätää ohjaamalla säätöportaita.
<b>stages</b>	Säätöportaat (taulukko)
<b>default</b>	Portaan oletusarvo, joka asetetaan portaan output kenttään kun säädin ei ole käynnissä.
<b>direction</b>	Säätöportaan suunta. Direct tarkoittaa että säätöportaan arvoa kasvatetaan kun mittausarvo on pienempi kuin asetusarvo. Inverted porras toimii päinvastoin.
<b>name</b>	Säätöportaan nimi (tiedoksi ihmisille)
<b>outMax</b>	Säätöportaan maksimiarvo.
<b>outMin</b>	Säätöportaan minimiarvo.
<b>output</b>	Säätöportaan tämän hetkinen oloarvo.
<b>outputId</b>	Tietokantapiste johon output arvo kirjoitetaan (voi jättää tyhjäksi ja lukea esimerkiksi ohjelmassa).
<b>pBand</b>	Säätöportaan suhdealue.

**hvacSchedule** Aikaohjelma, eli viikkokello johon voidaan liittää poikkeuspäiväkalenteri.

Viikkokellolle on ensiksi lähdön mahdolliset tilat. Tämä tehdään antamalla niille nimet **stateTexts** kentässä. Ne voivat olla esimerkiksi "pois" ja "päällä". Taulukon järjestyksessä ensimmäinen tila vastaa lähdön tilaa 0, seuraava rivi tilaa 1 jne. Sen jälkeen **weeklyEvents** kentän kautta voidaan lisätä tapahtumia viikon eri päiville vapaavalintainen määrä, joiden perusteella aikaohjelman **pv** kenttä vaihtaa arvoaan. Pisteiden **calendar** kenttään voidaan määrittää poikkeuspäiväkalenteri, jonka tapahtumat ajavat ylitse viikkokellon omista tapahtumista.

## hvacCalendar

Kalenteriohjelma, jota kutsutaan aikaohjelmien yhteydessä poikkeuspäiväkalenteriksi. Se on toisaalta tarkoitettu käytettäväksi aikaohjelmien yhteydessä, mutta sitä voidaan käyttää aivan hyvin myös ilman aikaohjelmaa lua ohjelmista käsin.

**stateText** kenttään tulisi ensiksi syöttää nimet kaikille kalenteriohjelman lähdön arvoille. sein riittää kaksi arvoa esim. pois ja päällä, tai seis ja käy. Sitten **dateList** kentän kautta voidaan lisätä ensiksi haluttuja ajanjaksoja kalenteriin, joiden mukaan kalenterin pv kentän arvoa vaihdetaan.

Tapahtuma kalenteri:

Jokaisella **dateList** taulukkoon lisätyllä jaksolla on seuraavat arvot.

**startDate** tarkoittaa mistä päivämäärästä jakso alkaa.

**duration** tarkoittaa montako päivää tätä jaksoa seurataan.

**event** on kyseisellä jaksolla noudatettavat päivittäiset tapahtumat (mihin aikaa lähtö menee päälle, ja milloin taas pois).

## hvacCurve

Vapaasti määriteltävä säätökäyrä, jonka **input** kenttään voidaan kirjoittaa lukuarvo, ja lukea sitä vastaava **points** kentässä annetun käyrän arvo **pv** kentästä. **y\_title** ja **x\_title** kenttiin voidaan antaa käyrän vastaavien akseleiden nimet, joka näytetään puolestaan käyttöliittymässä. Pistettä on mahdollista käyttää n.s. käsin lua-ohjelmassa, tai antaa **inputId** kenttään haluttu pistetunnus, jonka arvo viedään käyrän kautta hvacCurve-pisteen **pv** kenttään.

## hvacTimer

Viiveajastin, jota voidaan käyttää joko lua-ohjelmassa, tai suoraan pistetietokannassa.

Halutus viiveet määritellään pisteen onDelay ja offDelay kenttiin. Viiveiden yksikkö on millisekunti, joten arvo 1000 vastaa 1 sekuntia.

Tämän jälkeen, kun pisteen input kenttään kirjoitetaan jokin arvo, se siirtyy ajastimen output kenttään annettujen viiveiden jälkeen. Jos input arvossa tapahtuu pudotus (laskeva reuna) käytetään offDelay arvoa, ja jos input arvo kasvaa (nouseva reuna) käytetään onDelay arvoa. Pisteen **pv** kentästä voi lukea ajastimen kulloisenkin tilan: 0 tarkoittaa, että mikään viive ei ole kesken. 1 tarkoittaa että **offDelay** on kesken, ja arvo 2 tarkoittaa että **onDelay** juoksee.

Lisätoimintoja voidaan antaa **stepSize** parametri. Sen avulla määrittää kuinka suurissa askeleissa output kentän arvo muuttuu kohti **input** kentän arvoa jokaisen toteutuneen viivejakson jälkeen. Jos stepSize kenttään laittaa vaikkapa 0.2 tai 0.5 ja offDelay ja onDelay arvoiksi vaikkapa 1, saadaan ajastimella toteutettua liukumia kahden arvon välillä, koska output arvo muuttuu enintään stepSize kentän arvon verran yhdessä noin yhden sekunnin aikana.

# Ohjelmointityökalut

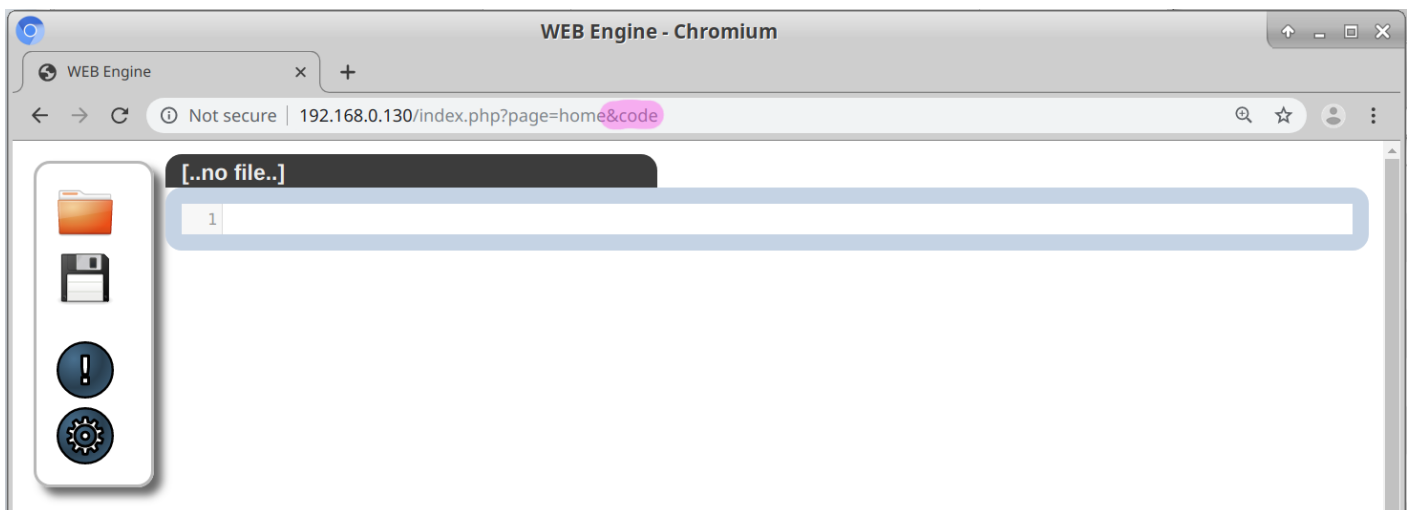
Actiweb järjestelmään on mahdollista tehdä sovellusohjelmia monella eri tavalla. Web käyttöliittymä sisältää pienimuoiden ohjelmaeditorin, ja Actiweb-laite on mahdollista liittää verkkolevynä windows tietokoneeseen, tai muokata niitä FTP-ohjelman kautta.

Ohjelmat sijaitsevat laitteessa tavallisesti lähdekooditiedostoina, eli tekstitiedostoina jotka sisältävät lua-kielistä ohjelmakoodia. Kirjastotiedostot suositellaan tallentamaan hakemistoon **/opt/slc/lib/**. Ohjelmakirjastolla tarkoitetaan lähdekooditiedostoa, joka sisältää sellaista uudelleenkäytettävää ohjelmakoodia, jota voidaan käyttää monessa eri projektissa. Muut, eli projekti tai sovelluskohtaiset ohjelmat on ajateltu sijoitettavaksi **/opt/slc/prg/** hakemistoon. Jako ei ole niinkään tekninen, vaan suositus, ja järjestelmä löytää ohjelmatiedostot ihan yhtä helposti molemmista sijainneista.

Ohjelmat, joiden halutaan käynnistyvän tehtäviksi – eli taskeiksi – sen sijaan pitää tallentaa **/opt/slc/prg/run/** hakemistoon.

## Ohjelmointi Web käyttöliittymän kautta

Lua-ohjelmien luominen ja muokkaaminen on mahdollista laitteen web-käyttöliittymän kautta. Ohjelmaeditoriin pääsee sisäänkirjautumisen jälkeen lisäämällä selaimen osoitekenttään laitteen URL-osoitteen perään parametri `&code`. Huomaa, että ohjelma editorin käyttäminen vaatii käyttäjätilin käyttäjätasoksi vähintään 100.



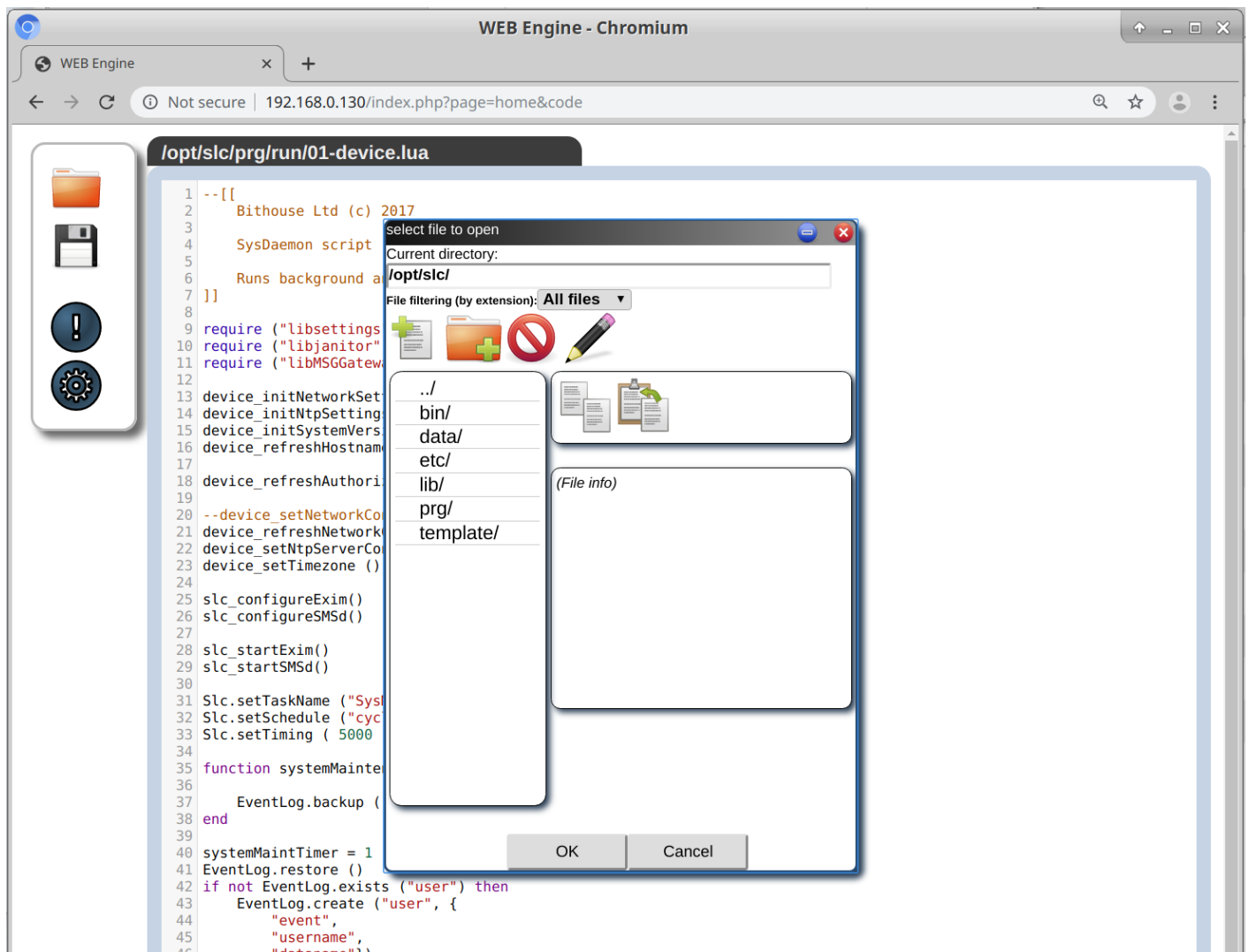
*Ohjelmaeditoriin siirrytään lisäämällä selaimen osoitekenttään &code URL-parametri*

Ohjelma editorin vasemmassa laidassa ovat päällekkäin painikkeet:

- Avaa (Kansion kuva) Avaa tiedoston laitteen levyltä.
- Save (Disketti) Tallentaa avoinna olevan tiedoston levyille.
- Viesti-ikkuna (Huutomerkki) Näyttää ohjelmien viestit ja virheilmoitukset

- Ohjelmalista (Hammasratas) Näyttää listan käynnissä olevista ohjelmista, sekä niiden tilat, suorituslaskurin ja mahdolliset Slc.echo() kutsulla annetut lisätiedot.

Ohjelman lähdekooditiedoston saa avattua muokattavaksi painamalla "avaa" -painiketta, ja valitsemalla avautuvasta ikkunasta haluttu tiedosto.



*Tiedostonvalintaikkuna ja taustalla muokattavaksi avattu Lua-kielinen ohjelma.*

Laitteessa käynnissä olevien ohjelmien tilaa on mahdollista seurata ikkunasta, joka avautuu ratas-painikkeesta. Avautuvassa listauksessa näkyvät riveittäin käynnissä olevat lua-ohjelmat, ja joitakin tietoja kustakin ohjelmasta.

PLC tasks				
Name	Rounds	Runtime	Status	Messages
alarmServer	17	404 ms	Running	0 alarm points found
dbServer	1	0 ms	Running	
SysManager	4	2487 ms	Running	Mem: 82.2958984375 kB
bacnetClient	21	1 ms	Running	Memory usage: 85.984375 kB
BacnetIP	60	251 ms	Running	Using 340.7060546875 kB
fileIO	29	360 ms	Running	
CURLio	5	268 ms	Running	
Fdx_client	20	0 ms	Running	0 of 0 offline
Mbus_master	4	391 ms	Running	0/0 meters online, 0 rx errors, Mem: 81.732421875 kB
ModbusMasterMain	2	0 ms	Running	
ModbusSlave	2619	51 ms	Running	Slave disabled
Reports	3	0 ms	Running	
Energy	1	1093 ms	Running	
FdxServers	816	14 ms	Running	
trendlogs	60	698 ms	Running	Checking buffers..
hvacServer	10	2734 ms	Running	
modbusTCP1	1548	48 ms	Running	Errors on last cycle: 0/0 reads, 0/0 writes
modbusPort1	1524	1 ms	Running	Errors on last cycle: 0/0 reads, 0/0 writes
Restart PLC			save DB	

Sarakkeiden kuvaukset:


**Name** Ohjelman nimi. voidaan vaihtaa Slc.setTaskName -kutsulla.

**Rounds** Suorituskertojen määrä, laskettuna Actiwebin edellisestä käynnistyksestä.

**Runtime** Aika, jonka taskin suorittaminen kesti edellisellä kerralla.

**Status** Taskin tila, voi olla loading, running tai fault.

Ohjelmaeditorissa huutomerkistä avautuva Output eli tuloste ikkuna näyttää laitteessa käynnissä olevien ohjelmien tulostamat viestit ja virheilmoitukset. Tähän ikkunaan tulevat näkyviin viestit jotka tulostetaan lua-ohjelmassa mm. yksinkertaisella **print()**-käskyllä (standardi tulostevirta) ja **Slc.error()** -käskyllä.



```
PLC output
Starting SLC Engine...
PLC version: 8.52.1625.969
Build version: 8.39.1358-217
Running default initialization script..
Loading default configuration file '/opt/slc/etc/config.lua'      ..OK
Searching autorun scripts.. found 15
** Switching to server mode (Wed Aug 21 12:50:43 2019) **
FdxServer: starting...
Setting timezone...
```

Itse ohjelman editointi onnistuu kuten normaalisit tekstieditoriohjelmissa. Kun muutokset ohjelmiin on tehty, pitää muokattu tiedosto tallentaa, ja painaa **TODO LAUSE LOPPUUN**



# libhvacex

Laajennettu hvac-kirjasto (hvac extras) sisältää hieman erikoistuneempia toimintoja joita kuitenkin tarvitaan usein kiinteistöautomaatiossa.

Kirjaston pistetyypit ladataan ja luodaan automaattisesti hvac kirjaston yhteydessä josse on asennettu.

Kirjaston käyttö:

## rajaohjaukset

Luo tietokantapiste käyttämällä **hvacLimitControl** schemaa.

Valitse **enabledId** kenttaan rajaohjausta ohjaava aikaohjelma. Jos ohjaukseen ei käytetä aikaohjelmaa, voi sen jattaa tyhjäksi jolloin ohjelma ohjaa rajaohjausta raja-arvojen ja vertailutyyppin mukaisesti.

**inputId** kentaan valitaan mittaus, jonka mittauksen mukaan ohjaus toimii.

**input** kentaan tulee inputId pisteen pv arvo automaattisesti näkyviin.

**outputId** kenttaan valitaan ohjattava piste.

**comparisonType** kentta on alasvetovalikko, josta voi valita millaisella ohjauksella rajaohjauksen halutaan toimivan.

**lowLimit** vaihtoehto laittaa ohjauksen paalle kun mittausarvo, joka tuodaan input kenttaan alittaa **controlLowLimit** kenttaan asetetun arvon.

Kun **highLimit** valittuna comparisonType valikosta, ohjaus menee paalle, kun mittausarvo ylittää **controlHighLimit** kentaan asetetun arvon.

Jos **between** vertailu tyyppi valittuna tarvitsee ohjelma sekä **compareHigh-** etta **compareLowLimit**. Tassa tapauksessa ohjelma ohjaa ohjauksen paalle, kun mittaus on rajojen valissa.

Rajojen läheisyydessä käytetään **hysteresis** kentän arvoa ohjausvälyksenä.

pv kentassa näkyy arvo johon ohjelma ajaa ohjattavaa pistettä.

## Ylärajahälytys (highLimitAlarm)

Luo hälytyspiste käyttämällä **alarm** schemaa ja luo sille uusi **highLimitId** kenttä. Kun tallennat tietokannan (save database) ja uudelleenkäynnistät (restart), ohjelma luo automaattisesti **inputId** kentän.

**inputId** kenttä on mittaus pisteelle, ja **highLimitId** on on ylärajalle tarkoitettu kenttä.

Ohjelma seuraa pisteiden arvoja automaattisesti, ja antaa hälytyksen kun **inputId** kenttään asetetun pisteen arvo ylittää **highLimitId** kenttään asetetun pisteen arvon.

Pisteeseen voi tarvittaessa luoda lisäksi valinnaiset kentät **hysteresis** ja **indicationId**, joihin voi antaa rajahälytyksen antamisessa käytettävän välyksen, ja pisteen josta luetaan valvotun prosessin käyntitila. Mikäli käyntitila saa arvon 0, raja-arvon ylitys ei aiheuta hälytystä.

### **Alarajahälytykset (lowLimitAlarm)**

Luo hälytyspiste käyttämällä **alarm** schemaa ja luo sille uusi **lowLimitId** kenttä. Toimii muutoin kuten ylärajahälytys, mutta antaa hälytyksen mikäli **inputId** kentässä annettu piste alittaa **lowLimitId** kentässä annetun pisteen arvon.

### **Saatovikahälytykset (errorMarginAlarm)**

Luo hälytys piste käyttämällä **alarm** schemaa ja luo sille uusi **errorMarginId** kenttä. Kun tallennat tietokannan(save database) ja uudelleenkäynnistät (restart), ohjelma luo automaattisesti **inputId** ja **setPointId** kentat.

**inputId** kenttä on mittauspisteelle jota valvotaan, **errorMarginId** pisteestä luetaan sallittu eroalue, jonka arvon verran mittaus voi erota asetusarvosta laukaisematta hälytystä (eli +/- eroalueen verran) ja **setPointId** on asetusarvolle tarkoitettu kenttä.

Ohjelma seuraa pisteiden arvoja automaattisesti, kun kentät on täytetty oikein ja oikeat pisteet ovat lisätyissä kentissä.

### **Ristiriitahälytykset (conflictAlarm)**

Luo hälytyspiste käyttämällä **alarm** schemaa ja luo sille uusi **controlId** kenttä. Kun tallennat tietokannan(save database) ja uudelleenkäynnistät (restart), ohjelma luo automaattisesti **inputId** kentän.

**inputId** kenttä on indikointipisteelle ja **controlId** kenttä on ohjaus pisteelle. Ohjelma seuraa pisteiden arvoja automaattisesti, kun kentat on täytetty oikein ja oikeat pisteet ovat lisätyissä kentissä.

Ristiriitahälytys annetaan mikäli **inputId** kentässä ja **controlId** kentässä asetettujen pisteiden arvot eroavat toisistaan. Vaihtoehtoisesti, jos **controlId** kenttään ei ole määritelty pistettä, hälytys

annetaan jos **inputId** kentässä oleva piste menee tilaan 0.

### **Kommunikaatiohälytys (commAlarmId)**

Lisäämällä hälytyspisteeseen kenttä **commAlarmId**, alkaa hvac-ex kirjasto valvoa kyseisessä kentässä annetun i/o-pisteen kommunikaation tilaa. Kun valvotun pisteen **commStatus** kentän arvo muuttuu pienemmäksi kuin 0 – eli valvottava piste menee offline tilaan – nousee hälytyspiste (johon commAlarmId kenttä on lisätty) hälytystilaan toAlarmDelay -viiveen jälkeen.

### **Käyntituntilaskenta (activeHours)**

Lisäämällä pisteeseen kenttä **activeHours**, alkaa hvac kirjasto laskea siihen tunteina kuinka kauan pisteen **pv** kenttä on ollut arvoltaan suurempi kuin 0. Tuntilaskurin saa nollattua kun tuntilaskuri ei ole aktiivinen, eli pisteen pv kenttän arvo on 0 (tai pienempi). Laskentatarkkuus on 1 sekunti.

# Pistetietokanta

Prosessidata tallennetaan CPU yksikön pistetietokantaan. Teknisesti tietokanta on suuri avain/arvo -taulukko, jonka sijaitsee erillisen käyttöjärjestelmäprosessin varaamassa muistissa.

Tietokannassa säilytetään dataa, jonka halutaan säilyvän järjestelmän uudelleenkäynnistyksen ylitse, näyttää käyttöliittymägrafiikassa, tai vaikkapa siirtää PLC ohjelmalta toiselle.

Tietokanta tallennetaan laitteen sisäiselle flash muistille **/opt/slc/data/persistent.json** -tiedostoon. Mikäli kyseistä tiedostoa ei ole laisinkaan olemassa kun laite käynnistyy (tai oikeammin slcengine ohjelma), luodaan laitteeseen tyhjä pistetietokanta. Mikäli tiedosto on korruptoitunut, eli se on olemassa, mutta siinä olevaa JSON dataa ei pystytä tulkitsemaan oikein, koettaa järjestelmä ladata ensin varmuuskopion **persistent.bak.1** ja mikäli se ei onnistu, sen jälkeenkoetetaan ladata toinen varmuuskopio **persistent.bak.2**. Jos mitään näistä tiedostoista ei saada ladattua, käynnistetään SLC Engine virhetilassa, jossa tietokannan levyoperaatiot ovat estettyjä. Tämä tehdään siksi, että ei ylikirjoitettaisi vahingossa persistent.json tiedostoa tai sen varmuuskopioita. Näin pistetietokannalle voi koettaa tehdä manuaalisen palautusoperaation mikäli edes osa datasta on edelleen lukukelpoista.

Pistetietokannan datapisteet voidaan ryhmitellä hierarkisesti hakemistoihin hieman tiedostojen tapaan, jossa hakemiston erottimena toimii kauttamerkki (/). Näin ollen prosessin ABC pisteet TE01, TE02 ja TE03 voidaan nimetä esimerkiksi ABC/TE01, ABC/TE02 ja ABC/TE03. Tämän jälkeen prosessin ABC -tietokantapisteet on mahdollista hakea vaikkapa komennolla **Data.list ("ABC/\*", "")**. Toisaalta, myös tietokantaselain näyttää kaikki kyseisen prosessin pisteet käyttöliittymässä samassa hakemistossa.

Tietokantapisteen nimi voi koostua merkeistä **a..z, A..Z, 0..9, / ja \_**. Muita merkkejä ei voi käyttää, sillä pisteiden luontiin käytetty kutsu Data.create () epäonnistuu mikäli pisteen nimi sisältää kiellettyjä merkkejä.

Tietokantapisteitä on mahdollista käsitellä Plc ohjelmassa "Data" -kirjaston kutsuilla. Kirjastossa on kutsut joilla on mahdollista paitsi lukea ja kirjoittaa pisteitä, myös listata, luoda ja poistaa niitä.

Tietokannan datapiste koostuu niinsanotuista kentistä (eng. field) – joita kutsutaan myös tietueksi tai ominaisuuksiksi (properties). Kyse on kuitenkin samasta asiasta. Ja tietokantapisteen tieto onkin varastoitu juuri näihin kenttiin, esimerkiksi pisteen ABC/TE01 kenttä 'pv' voi sisältää lämpötilamittauksen varsinaisen mittausrvon.

Kenttien nimiä koskevat samat nimeämissäännöt kuin itse pisteitä, paitsi että myös '/' -merkki on kielletty. Kentät voivat sisältää paitsi yksinkertaisia luku tai tekstiarvoja, mutta myös monimutkaisempia tietorakenteita, kuten taulukoita (object) tai listoja (array). Tällaiset monimutkaisia datarakenteita sisältävät pisteet on kuitenkin luotava n.s. mallin (schema) pohjalta.

Tietokantapisteillä voi siis olla myös ominaisuus jota kutsutaan malliksi. Se määrittää tarkalleen mitä kenttiä piste voi sisältää, ja millaista tietoa kukin näistä kentistä voi pitää sisällään. Kyse on samasta asiasta kuin tietokanta tai XML skeemojen kohdalla. Myös näitä skeemoja, eli malleja on mahdollista luoda Plc ohjelmissa "Data" kirjaston kutsuilla.

Tietokanta web käyttöliittymässä.

Kun käyttäjä luo uutta pistettä käyttöliittymän avulla, ja painaa "+ New" painiketta tietokantaeditorissa, häneltä kysytään avautuvassa dialogissa paitsi pisteen nimeä, myös sen tyyppiä. Tuo pisteen tyyppi tarkoittaa nimenomaan mihin malliin (eng. schema) piste sidotaan. Pisteiden mallit voivat esimerkiksi määrätä, että pisteessä on oltava kentät description, pv ja dataSource. Lisäksi malli määrittää mitä datatyyppiä kentät ovat.

Datapisteiden kenttien tyypit:

**boolean** (aliakset: bool)

Totuusarvo, eli voi saada arvon "true" tai "false"

**real** (aliakset: float, number)

Reaaliluku, eli tutummin desimaaliluku. Tarkkuus on aina 64-bit.

**integer** (aliakset: int)

Kokonaisluku. Monissa yhteyksissä Plc ohjelmissa tämä myös tätä tyyppiä käsitellään 64-bittisenä liukulukuna, mutta aina kun se tallennetaan pistetietokantaan, se muutetaan 32-bittiseksi kokonaisluvuksi.

**string** (aliakset: text)

Merkkijono, eli teksti. Tietotyyppillä on oikeastaan vain yksi huomion arvoinen rajoite; Se ei voi sisältää n.s. kaksois-hipsua, eli lainausmerkkiä (""). Sallittu enimmäiskoko on >4KBi (eli 4096 ASCII merkkiä).

**reference**

Viittaus toiseen tietokantapisteeseen. Oikeastaan kyse on teksti tyyppisestä datasta, ja tärkein ero string -tyyppiseen tietueeseen onkin siinä että sille voidaan näyttää erilainen käyttäjän syöteikkuna.

**array**

Lista muotoinen tietue. Listan ja taulukon suurin ero on siinä, että listalla on aina järjestys, ja listan tietueet löytyvät peräkkäisistä kokonaisluvulla osoitettavista soluista. Listan ensimmäinen solu on aina indeksissä 1, ja jos ensimmäinen solu poistetaan, muuttuu kaikkien jäljelle jäävien solujen indeksi yhdellä pienemmäksi. Lisäksi numerointi on aina juokseva, eikä välissä ole koskaan aukkoja.

Plc ohjelmissa array tyyppinen kenttä näkyy lua taulukkona, jolla on juokseva numeraalien indeksi. Lua tutoriaaleissa siihen viitataan array -tyyppisenä indeksointina.

**object**

Tämä datatyyppi on toisaalta melkotavalla samanlainen kuin lista, mutta numeraalisten indeksien sijasta sen soluihin osoitetaankin nimillä, hieman samaan tapaan kuin itse tietokantapisteiden kenttiin. Tämä aiheuttaa sen, että objektin soluilla ei ole mitään suurempaa sisäistä järjestystä, jossa yhden voisi sanoa tulevan ennen toista. Toisaalta, ne eivät myöskään vaihda paikkaansa kun jokin soluista poistetaan. Object tyyppisen kentän solut voivat olla nimeltään vaikkapa "maanantai", "tiistai" ja "keskiviikko".

Plc ohjelmissa object tyyppinen kenttä näkyy lua taulukkona, jonka soluilla on "string" tyyppiset indeksit, ja joita voidaan listata esimerkiksi pairs() -funktiolla.

Datapiste voidaan lukea Plc ohjelmassa funktioilla:

```
Data.get ("point_id")  
Data.getInt ("point_id")  
Data.getReal("point_id")  
Data.getString("point_id")
```

Funktiot toimivat samalla tavalla, erona on vain paluuarvon tyyppi. Data.get () voi palauttaa minkä tyyppisen arvon tahansa, ja sde on ainoa tapa lukea **array** tai **object** muotoinen kenttä.

Data.getInt() palauttaa aina kokonaisluvun, Data.getReal() palauttaa aina desimaaliluvun, ja Data.getString () puolestaan merkkijonon - niinkun nimikin kertoo.

On tärkeätä huomata, että näiden kutsujen erot ovat nimenomaan virhetilanteissa! Mikäli luettava tietue on olemassa, ja sen tyyppi on määritelty skeemassa "real", palauttaa myös Data.get() liukuluvun. Data.getString () puolestaan muuttaa datan merkkijonoksi, ja palauttaa sen. Data.getInt () leikkaa desimaalit pois, ja palauttaa kokonaislukuosan. Data.getReal () taas pakottaa luvun liukuluvuksi (eli tässä tapauksessa ei tee sille mitään) ja palauttaa sitten arvon.

Tällä on merkitystä Plc ohjelmien yksinkertaisuuden ja virhesietoisuuden kanssa. Mikäli pyydetty tietue ei olekaan ohjelmoijan ennakoimaa tyyppiä, tai se on poistettu, palauttavat Data.getInt() ja Data.getReal() aina arvoksi 0. Data.getString () palauttaa virhetilanteessa tyhjän merkkijonon – mutta kuitenkin merkkijonon. Dat.get () puolestaan voi palauttaa arvon **nil**, mikäli pyydettyä tietokantapistettä ei ole olemassa.

Kun käyttää datatyyppiin sopivaa aliasta Data.get() -funktiolle ohjelmaa kirjoittaessa, voidaan työläästä tyyppitarkistukset jättää usein tekemättä ohjelman toimivuuden kärsimättä.

Pisteitä on mahdollista lukea Data.get() kutsulla enemmän kuin yksi! Pisteiden nimessä on mahdollista käyttää \* merkkiä n.s. jokerina, jolloin ensimmäisestä argumentista muodostuu n.s. maski. Kutsu palauttaa taulukkona kaikki sellaiset pisteet, joiden nimi sopii annettuun maskiin.

Datapiste kirjoitetaan Plc ohjelmassa:

```
Data.set ("point_id", value)
```

Kutsun käyttäminen sinänsä on hyvin suoraviivaista, esimerkiksi pisteen ABC/TE01 kenttä pv voidaan kirjoittaa arvoon 19.4 lua-komennolla

```
Data.set ("ABC/TE01.pv", 19.4)
```

Toisaalta, kutsua voidaan käyttää myös toisella tavalla, jossa yhteen pisteeseen on yhdellä kutsulla mahdollista kirjoittaa useampi kenttä:

```
Data.set ("ABC/TE01", {pv=19.4, commStatus=1})
```

Ylläoleva esimerkki vaikuttaa varmasti lua-kieltä tuntevalle melko selvältä, eli toisena argumenttina voidaan antaa yksittäisen lukuarvon tai merkkijonon sijasta taulukko, jonka alkiot kutsu kirjoittaa pisteen kenttien arvoiksi. Ylläolevassa esimerkissä pisteeseen kirjoitetaan pv kenttään 19.4 ja commStatus kenttään arvo 1.

# Tietokantapisteiden kentät

Lähtökohtaisesti ohjelmisto ei määrää sitä, minkä nimisiä ja minkä muotoisia kenttiä tietokantapisteeseen voi luoda. On kuitenkin olemassa joitakin kenttien nimiä, jotka on varattu ohjelmiston sisäisesti tiettyyn tarkoitukseen. Alla on näistä varatuista, erityis merkityksen omaavista kentistä.

Huomaa, että lista ei välttämättä ole täydellinen, ja jotkin kolmannen osapuolen/erityiskentät kirjastot voivat tehdä oletuksia, joita tässä ei ole mainittu.

## **description (string)**

Järjestelmä olettaa tämän kentän olevan pisteen vapaamuotoinen kuvausteksti.

## **pv (vaihtelee pistetyypeittäin)**

Järjestelmä olettaa tämän kentän olevan pisteen varsinainen datasisältö, eli oloarvo. Sen datatyyppi voi vaihdella sen mukaan, mikä piste on kyseessä. AV pisteellä se on real tyyppinen, kun taas HVAC kirjaston PID säätimellä se on taulukko, joka sisältää real -tyyppisiä arvoja (yhden per säätöporras).

## **priority (int)**

Ilmoittaa millä prioriteetilla pisteen oloarvo on kirjoitettu. Prioriteetit vastaavat BACnet standardin prioriteetteja, eli 16 on pienin (n.s. auto tila) prioriteetti, ja 8 vastaa käsiohjausta (manual tila). Suuremman prioriteetin kirjoitus hylätään, ja 16 on vähäisin, ja 1 suurin sallittu prioriteetti.

## **writable (int)**

Määrittää, voiko pisteeseen kirjoittaa BACnet väylän kautta. Arvo 1 tarkoittaa, että kirjoittaminen on sallittua.

## **dataSource (string)**

Tämä kenttä sisältää tiedon siitä, mistä pisteen oloarvo tulisi lukea. Tarkemmat kuvaukset kommunikaatio protokollien yhteydessä.

## **dataTarget (string)**

Tämä kenttä sisältää tiedon siitä, mihin pisteen oloarvo tulee kirjoittaa. Tarkemmat kuvaukset kommunikaatio protokollien yhteydessä.

## **commStatus (int)**

Ilmoittaa, onko piste online vai offline tilassa. Arvo 1 tarkoittaa online ja/tai OK tilaa, ja sitä pienemmän erilaisia mm. kommunikaatio protokollasta riippuvia häiriötiloja. Tämä liittyy esimerkiksi dataSource tai dataTarget kentissä määriteltyyn väylä- tai lähiverkko liikenteeseen.

## **commTxt (int)**

Teksti muotoinen kuvaus siitä, millainen häiriö pisteen kommunikaatiossa on. Mikäli pisteen kommunikaatio ei ole häiriöllä, tekstin tilisi sisältää "Online" -merkkijono (suositus).



### **unit (int)**

Pisteen BACnet yksikkö, numeraalinen arvo tulee BACnet standardista.

### **dispUnit (string)**

Tekstimuotoinen yksikkö joka näytetään joissakin grafiikka elementeissä (kaikki eivät tue).

### **lowLimit(number)**

Alaraja pv kentän numeeriselle arvolle, eli tämän pienempää numeerista arvoa ei ole mahdollista kirjoittaa pisteen pv kenttään.

### **highLimit (number)**

Yläaraja pv kentän numeeriselle arvolle, eli tämän suurempaa numeerista arvoa ei ole mahdollista kirjoittaa pisteen pv kenttään.

### **onDelay (number)**

Tämä kenttä luo viivettä siihen, kuinka nopeasti pisteen raw arvon nouseva reuna välittyy oloarvoon, eli tavallisesti vaikutus on sama kuin vetohidastuksella. Toimii vain silloin kun pisteen arvo luetaan tai kirjoitetaan jonkin kommunikaatorajapinnan kautta. Yksikkö on millisekunti.

### **offDelay (number)**

Tämä kenttä luo viivettä siihen, kuinka nopeasti pisteen raw arvon laskeva reuna välittyy oloarvoon, eli tavallisesti vaikutus on sama kuin päästöhidastuksella. Toimii vain silloin kun pisteen arvo luetaan tai kirjoitetaan jonkin kommunikaatorajapinnan kautta. Yksikkö on millisekunti.

### **curve (reference)**

HVAC kirjaston toteuttama ominaisuus. Toimii dataSource kentän yhteydessä, ja tähän voidaan antaa pistetunnus, joka viittaa HVAC kirjaston hvacCurve pisteeseen. Tämän jälkeen kyseistä hvacCurve-käyrää käytetään kun raw arvo skaalataan pisteen oloarvoksi (pv-kenttä).

## **Data -kirjasto: rajapinta pistetietokantaan**

### **Data.create ("pointId" [, "schema"], initialValues )**

Tällä funktiokutsulla luodaan uusia datapisteitä tietokantaan. Kutsu palauttaa *true* mikäli pisteen luominen onnistui, ja *nil* mikäli se epäonnistui. Mikäli samanniminen piste on jo olemassa, kutsu palauttaa virheen (eli *nil*) eikä muuta millään tavalla olemassa olevaa pistettä.

Uudet kentät olemassa oleviin pisteisiin luodaan Data.set () kutsulla.

pointId on luotavan pisteen nimi. Sallitut merkit ovat pcre syntaksilla ilmaistuna [a..zA..Z0..9/\_].

schema parametri on valinnainen, ja kertoo luotavan pisteen tyyppin, eli skeeman.

initialValues on taulukko, jossa voidaan antaa luotavan pisteen kenttien arvot, esimerkiksi halutut hälytysviiveet, kuvaus, tai muita tietoja.

```
-- Create new point, and attach it to schema 'AI'
```

```
Data.create("ioPoints/TK01TE10_AI", "AI", {description="Inlet air temperature"})
```

```
-- Create new point, do not attach it to any schema
```

```
Data.create("point/with/no/schema", {description="I has no schema"})
```

**Data.get ("pointId" [, "strFilter"] )**  
**Data.getInt ("pointId" [, "strFilter"] )**  
**Data.getInteger ("pointId" [, "strFilter"] )**  
**Data.getInteger ("pointId" [, "strFilter"] )**  
**Data.getReal ("pointId" [, "strFilter"] )**  
**Data.getNumber("pointId" [, "strFilter"] )**  
**Data.getString ("pointId" [, "strFilter"] )**

Tällä funktiokutsulla luetaan varsinaisten datapisteiden arvoja pistetietokannasta. Se palauttaa joko pyydetyn pisteen tai kentän arvon, tai *nil* mikäli arvoa ei löydy.

*Data.get()* funktion variaatiot pakottavat paluuarvon tiettyyn tyyppiin, joka voi yksinkertaistaa ohjelmaa myöhemmin. Perusmuotoinen *Data.get ()* voi palauttaa mitä datatyyppiä tahansa, ja palauttaa arvon *nil* mikäli haettua tietokantapistettä ja/tai kenttää ei ole olemassa. *Data.getString()* palauttaa tyhjän, 0 pituisen merkkijono, ja *Data.getInt()* – ja muut numeraaliset arvon palauttavat funktiot – antavat luvun 0 mikäli haluttua tietoa ei ole olemassa (tai se on väärää muotoa).

*pointId* parametri voi olla joko pisteen nimi – jolloin kutsu palauttaa koko pisteen taulukkona, tai viittaus johonkin pisteen kenttään, jolloin kutsu palauttaa vain tuon kyseisen kentän arvon.

Kyseinen parametri voi olla jopa wild card -merkkejä sisältävä suodatin, jolloin yksi *Data.get ()* kutsu palauttaa kaikki ne pisteet joihin suodatin täsmää. Tässä moodissa on mahdollista käyttää myös *strFilter* parametria, jolla voidaan tehdä suodatusta pisteen sisältämien kenttien perusteella.

*Data.get("ABC/TE01") -- Return whole datapoint as table (or nil on failure)*

*Data.get("ABC/\*") -- Get all points starting with 'ABC/'*

*Data.get("/\*", "(pv > 0)") -- Returns all points having pv greater than zero.*

### **Data.set ("pointId", value)**

Asettaa datapisteen arvon. Tällä menetelmällä voidaan asettaa joko yksittäisen kentä arvo, tai useamman kentän arvo yhdellä kutsulla. Tästä kutsusta ei ole olemassa useampaa variaatiota, vaan tietokantaan kirjoitettava arvo on muutettava haluttuun muotoon ennen kutsun suorittamista. *pointId* on merkkijono joka kertoo mihin tietokantapisteeseen halutaan kirjoittaa. Se voi olla joko pisteen nimi, tai sisältää pisteellä erotettuna myös tietokantapisteen kentän.

*value* on tietokantapisteeseen kirjoitettava arvo. Tässä on olemassa kaksi mahdollisuutta. Mikäli kirjoitetaan vain tiettyyn tietokantapisteen kenttään, tämä parametri on usein numero tai merkkijono. Toisaalta, mikäli kirjoitus kohdistuu koko tietokantapisteeseen, *value:n* tulee olla taulukko, joka kirjoitetaan pisteeseen niin, että taulukon rivi "pv" kirjoitetaan pisteen vastaavaan kenttään "pv". tällä tavalla voidaan kirjoittaa tietokantapisteen monta kenttää yhdellä kutsulla. Kutsu palauttaa true mikäli kirjoitus onnistui, tai nil mikäli se epäonnistui. Kutsu voi myös luoda uusia data-kenttiä pisteisiin, mikäli datapisteen skeema sen sallii.

*Data.set ("ABC/TE01.pv", 19.4)*

*Data.set ("ABC/TE01", {pv=19.4, commStatus=1, commTxt="Online"})*

### **Data.list ("pointIdFilter"[, "fieldFilter"])**

Tällä kirjastokutsulla on mahdollista listata tietokantapisteitä haluttujen ehtojen mukaisesti. Kutsu palauttaa onnistuessaan taulukon – tyhjän taulukon mikäli yhtään ehtoihin täsmäävää pistettä ei löytynyt, tai nil mikäli kutsu epäonnistui.

*pointFilterId* on pisteiden nimiin sovellettava suodatin, jossa voi käyttää wild card merkkeinä \* ja ? merkkejä.

*fieldFilter* on puolestaan lua kielinen vertailu, jossa voi testata esimerkiksi pisteen kenttien olemassaoloa ja/tai arvoja. Tälle lyhyelle skriptille annetaan paikallisina muuttujina kaikki testattavan pisteen kentät, joten se voi testata niitä hyvin helposti. Pisteeseen katsotaan läpäisseen testin, jos tämä skripti palauttaa jonkin muun arvon kuin *false* tai *nil*.

Huomaa, että tämä funktio palauttaa suodattimiin täsmäävien pisteiden nimet listana, ei pisteiden arvoja. *Data.get()* puolestaan, mikäli sitä kutsutaan suodattimien kanssa, palauttaa pisteiden arvot. Sen vuoksi suorituskyvyn kannalta tämä kutsu on huomattavasti nopeampi.

```
local t = Data.list ("ioPoints/TK01*")  -- list all point starting with 'ioPoints/TK01'
local t = Data.list ("*")              -- list all points
local t = Data.list ("*", "(pv > 0)")    -- list all points with pv greater than zero
```

### **Data.remove("pointId", "fieldFilter")**

Poistaa tietokantapisteen, tai pisteitä. Palauttaa poistettujen pisteiden määrän numerona, tai *nil* jos kutsu epäonnistui.

*pointId* on poistettavan pisteen nimi, tai samanlainen \* ja ? merkkejä sisältävä nimisuodatin kuin esimerkiksi *Data.list()* kutsulla.

*fieldFilter* toimii kuten *Data.list()* kutsulla.

```
Data.remove ("ioPoints/TK01/TE01_AI")  -- Remove single point
Data.remove ("ioPoints/TK01*")         -- Remove all points beginning with 'ioPoints/TK01'
Data.remove ("*", "(dataSource)")      -- Remove all points having dataSource field
```

### **Data.clone("pointSrc", "pointDst")**

Tällä kutsulla on mahdollista monistaa jo olemassa oleva tietokantapiste. Kutsu luo uuden pisteen nimellä *pointDst* joka on pisteen *pointSrc* täydellinen kopio kutsuhetkellä.

Kutsu palauttaa *true* jos monistaminen onnistui, tai *nil* mikäli se epäonnistui.

```
-- Creates new point called ' ioPoints/TK01/TE31 '
-- from point ' ioPoints/TK01/TE30 '
Data.clone ("ioPoints/TK01/TE30", "ioPoints/TK01/TE31")
```

### **Data.exists("pointId")**

Suorittaa testin onko *pointId* niminen piste tietokannassa. Funktio palauttaa *true* jos piste on olemassa, tai *false* mikäli sitä ei löydy.

```
if Data.exists ("ioPoints/TK01/TE10") then
    -- Point exists, do something
```

*else*

*-- Point does not exist, act properly*

*end*

### **Data.count ("pointIdFilter" [, "fieldFilter"])**

Laskee *pointIdFilter* ja valinnaiseen *fieldFilter* suodattimeen täsmäävien pisteiden määrän tietokannassa. Palauttaa numeron jos kutsu onnistui, ja nil jos se ei onnistunut. Huomaa, että se ei ole virhe jos tietokannasta ei löydy suodattimeen täsmääviä pisteitä, vaan kutsu palauttaa numeron 0 (nolla). Kutsu voi epäonnistua esimerkiksi siksi, että suodattimet sisältävät kiellettyjä merkkejä.

*-- Counts active alarms*

*Data.count("\*", "(av and av > 0)" )*

# OPC-UA

## Client

Kommunikointi OPC-UA palvelimeen tapahtuu opcua\_client.py -ohjelman kautta. Ohjelma seuraa pisteitä, joiden dataSource- tai dataTarget -kenttä alkaa "OPC://" -tagilla. Kentällä dataSource piste lisätään seurattavien(subscribe) listalle, ja dataTarget kentällä pisteen arvoa lähetetään määrävälein OPC palvelimelle.

### OPC -osoite

dataSource/dataTarget

1. OPC://{URL}/{ID}
2. OPC://{URL}/{BROWSE\_NAME1}/{BROWSE\_NAME2}...

Kaarisulkeilla merkityt parametrit annetaan URL encoded -muodossa.

{URL} : Paikalle tulee (IP tai domain):portti.

{ID} ja {BROWSE\_NAME} : Ulkoisella OPC -palvelimella olevan pisteen(node) voi antaa joko "nodeid":n tai "browse name":n perusteella. {ID} on uniikki id, joka on usein muotoa "ns=2;s=xxx". {BROWSE\_NAME} lähtee selaamaan palvelinta kuten tiedostokansioita, ja selaaminen aloitetaan palvelimen "Objects" polusta. Nämä annetaan muodossa "0:xxx".

### Pisteiden arvot ja bitmask

Actiweb tietokannan opc client -pisteillä on arvot "pv" ja "raw". Ensimmäinen on tarkoitettu Actiweb sovellusohjelmien ja web UI:n käyttöön, ja jälkimmäinen luetaan/kirjoitetaan opcua\_client -ohjelmasta. Actiwebin sovellus OPC\_client tekee siirron näiden välillä, ja tekee tarvittaessa samalla bittimaskin tarvitsemat toimenpiteet.

Luettaessa bitmask pistettä, UInt16 arvo tallennetaan normaalisti "raw" -kenttään. Mikäli pisteelle on annettu "targetList" -kenttään ei-tyhjä lista, UInt16 hajotetaan bit array -muuttujaksi. Sen alkiot jaetaan "pv" -arvoiksi "targetList" -kentän määräämille pisteille. Jakaminen perustuu listan pisteiden "bit" -kenttien arvoihin, jotka määräävät bit array -muuttujan indeksin.

Kirjoitettaessa maskia OPC palvelimelle, järjestys on päinvastainen. "targetList" -ketän määäämien pisteiden "pv" -kenttien arvot kootaan bit array -muotoon, muutetaan UInt16 -muuttujaksi, ja tallennetaan opc -pisteen "raw" -kenttään.

## Endpoint

Opcua\_client -ohjelman palvelinkommunikaatio parametroidaan "opc\_endpoint" -tyyppisellä tietokantapisteellä. Sen "endpoint" -kenttään annetaan täysin sama osoite, kuin "dataSource/dataTarget" -kentissä {URL} tagin tilalla on systemaattisesti käytetty, mutta ilman URL encode muunnosta. Kentän "interval" -arvoa(ms) käytetään subscribe -toiminnon tarkkuutena. Muut kentät toteuttavat OPC-UA standardin mukaiset kirjautumisvaihtoehdot. Käyttäjätunnus, salasana, ja certit tallennetaan tiedostoihin, ja tietokantaan tallennetaan ainoastaan niiden polut.

# Server

OPC UA palvelin pyörii opcua.py -ohjelman kautta. Ohjelma luo OPC UA datapisteet Actiwebin tietokantapisteistä, joiden tyyppi on AI, AO, AV, BI, BO, BV, MSI tai MSV. Ohjelma hakee pisteiden arvot viiden sekunnin välein ja päivittää oman tietokantansa arvot, jos ne ovat muuttuneet. Jos arvot palvelimen tietokannassa muuttuvat asiakasohjelman (client) muuttamana, kirjoittaa palvelin muutokset Actiwebin tietokantaan.

Palvelin luo itselleen **endpointin:** **opc.tcp://0.0.0.0:4840/freeopcua/server**

sekä rekisteröi **URI:n:** **"0.0.0.0:4840/freeopcua/server"** namespace ID:ksi.

Tietokantapisteiden nimet palvelin muuttaa siten, että kauttaviivat ( / ) muuttuvat väliviivoiksi ( - ).

Esim.: Actiwebissa **"energiaMittari1/teho.pv"** <-> OPC UA:ssa **"ns=2;s=energiaMittari1-teho.pv"**

Korvaavan merkin, endpointin ja URI:n voi tarvittaessa muuttaa opcua.py -tiedostoa tekstieditorilla muokkaamalla.

Tällä hetkellä palvelimeen ei ole toteutettu kirjautumismahdollisuutta.

# 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 samaan 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.

## Tärkeää 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 sovellusohjelman tarvitse välittää asiasta. Oikea tapa käyttää näitä funktioita 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 millisekunteinä

### **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 "periodic" 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

Funktiolla 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 pääfunktia 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 suoritettua lua-koodin paluuarvo takaisin ohjelmaan.

Funktiota voidaan käyttää hyvin monipuolisesti esimerkiksi tiedonsiirtomenetelmänä.

**Huomioi!** Tämä protokolla ei sisällä laisinkaan tietosuojaminäisyyksia! 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 käyttöjärjestelmän 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 ensimmäistä 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.

Esimerkinä:  
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.parseUri ( urlString )  
if parsed.protocol == "http" then  
  -- Handle http request  
end
```

**System.encodeUri (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 dekodeaus.

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.

Tämä 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 puskuuriin kirjoitetaan (huom! Arvo muutetaan BYTE tyyppiseksi, lukualue 0x0 .. 0xFF)  
*i* Indeksi josta täyttö aloitetaan (oltava  $\geq 0$ )  
*n* Kirjoitettavien tavujen määrä (puskurin indeksistä *i* lähtien).

Kutsu kirjoittaa BitOps.createBuffer -kutsulla luotuun muistipuskuriin 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 kopiaimalla vanha sisältö uuteen – jos uusi puskuuri on pienempi kuin vanha, kopioidaan vain se osa joka uuteen puskuuriin mahtuu.

Esimerkki:

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

### **BitOps.getBytes (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 indeksointi seuraa lua käytäntöä, eli puskurin ensimmäinen tavu on indeksissä 1!

Esimerkki:

```
-- Luo puskuuri  
local buf = BitOps.createBuffer (2048)  
BitOps.setString(buf, 1, "Handling buffer")  
local byte = BitOps.getBytes(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 muuttaa 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* Indeksiksi 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 tyyppissä.

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 tapauksessa 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-aulukon XML muotoon.

Huomaa, että lua-aulukon 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)
```

# alarm aux

**Hakee annetusta sähköpostilaatikosta hälytysviestejä osaksi Actiweb järjestelmää**

## Käyttöönotto

Paketti asennetaan kopioimalla src -hakemisto aplikaatioksi. Tarkemmat tekniset ohjeet löytyvät app.info -tiedostosta.

Tämä aplikaatio on tarkoitettu yleiseksi ratkaisuksi, ja siksi se vaatii kohdekohtaisen parsintaskriptin polkuun "/opt/slc/bin/handlers/{SOURCE}.py". "{SOURCE}:{EMAIL}" -pari , jossa {EMAIL} on hälytysten lähettäjän osoite tai sen osa, lisätään alarmsaux.config -tiedoston kenttään "handlers". Tätä tiedostoa pääsee muokkaamaan myös sivulla "system/alarms\_aux".

Edellä mainittuun tiedostoon konfiguroidaan myös sähköpostin IMAP kirjautumisasetukset.

Kohdekohtaisessa parsintaskriptissä tulee olla parse -niminen funktio, jolle annetaan parametreinä sähköpostin sisältö, ja ulkoisten hälytysten tiedostopolku. Skriptin tulee lisätä polkuun kuvaava alakansio ja tallentaa viestin sisällöstä parsittu hälytys sinne tiedostona. Sovellus vaatii riittävän uuden bitware version, koska ulkoiset hälytykset luetaan libalarmsaux.lua tiedoston funktioilla. "Parse" palauttaa tekstimuodossa rivin, joka tallennetaan tapahtumalogiin.

## Käyttö

Kaikki konfiguroituun sähköpostiosoitteeseen lähetetyt viestit haetaan ulkoisiksi hälytyksiksi järjestelmään. Ulkopuoliset hälytykset eivät tallennu tietokantaan, vaan JSON objekteina omiin tiedostoihinsa /opt/slc/data/alarms/{SOURCE}. Mikäli lähettäjän sähköpostiosoite ei vastaa yhtäkään parsintaskriptiä(handlers), tallennetaan se parsimattomana alahakemistoon "default".

Ulkoiset hälytykset näkyvät hälytyslistassa ja -indikaattorissa muuten normaalisti, mutta listassa niitä ei voi kuitata, vaan poistaa. Poistaminen tarkoittaa tiedoston poistamista levyltä, eikä hälytyksen lähettäjälle ilmoiteta kuittauksesta. Toiminto on rakennettu sillä oletuksella, että hälytysvalvomoon lähetetään poistokomento, kun tilanne on normalisoitunut, ja hälytyslistan "poista" -nappi on virhetilanteita varten. Ulkoiselle hälytykselle voi antaa "alarmGroup" -kentän, joka näkyy listassa, mutta jota ei voi linkittää tietokannan "alarmGroup" -tyyppisiin pisteisiin.