

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 merkistötaulukon 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 tehtään 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 tyyplitettyihin 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 tyypitetty 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 suoritussy järjestykseen, 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. suoritusjärjestyksestä johtuvat piirteet ovat, että *and* operaatio palauttaa arvon, joka annetaan sen oikealla puolella (eli järjestyksessä jälkimmäisen), jos molemmat puolet ovat totussarvoltaan *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* opertaation 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>

Revision #8

Created 24 May 2022 13:03:10 by Severi Hiltunen

Updated 10 May 2023 10:25:34 by Severi Hiltunen