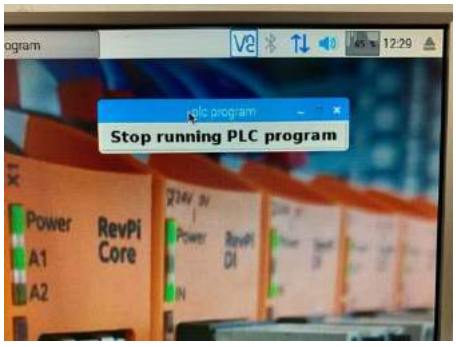
Smart Industry – Zelf aan de Slag (SIZAS-RevPi)



The work was made possible by Min EZK, provincie NoordBrabant and TNO.
This material is planned to submitted by the Smart Industry program (www.smartindustry.nl) into open source.

0 Preparation

Pre 1: needed devices, software & network

You need a notebook with MS-Windows (or MacOS or Ubuntu) to run *vncviewer, revpipycontrol* and you need an installation of *Python3*.

Note that your notebooks must allow the user to install software. Some company notebooks do not allow user to install software. In that case ask for a travel computer or use your private machine.

A Revolution Pi from Kunbus (short RevPi) is provided by the course. This is a modern industrialized Pi with 24VDC I/O modules (www.revolution.kunbus.com (or .de))

For the MQTT assignment with the MQTT.fx tool one needs a broker. The course will provide it on a Pi used as an edge computer to the RevPi, but you can run it self too.

Wi-Fi network name: smart
Wi-Fi network password: industry
10.0.0.0 is the Wi-Fi network address range with 10.0.0.x/24 (i.e. 10.0.0.1-10.0.0.255)
10.0.0.1 is the gateway
10.0.0.2 is Rev-2, a RevPi core-1 (participant experiment machine, user smart, …)
-.-.-.3 is Rev-3, a RevPi core-3 (demo machine, not to be used by participants)
10.0.0.4 is Pi-4, the Pi (model 4) with the demo I/O and OPC Server
10.0.0.11-16, are the Pi-11 to Pi-16, your course Pi's (these are Pi model 3 b+)
10.0.0.20-100 are IP addresses for the notebook computers in the smart Wi-Fi network

You need the following software, either downloaded from the Internet or the USB stick. It will run on a Microsoft Windows 10, Apple Mac OS and Unix Ubuntu computer.

python3 from www.python.org with the revpimodio2 library

vncviewer.exe from realvnc.com and (opt.) putty.exe from www.putty.org

https://github.com/ejsol/Smart-Industry-zelf-aan-de-slag.git
for the Python programs with the Pi + Grove and the Python programs with the RevPi

MQTT.fx from https://mqttfx.org (+ java from www.java.com) for observing MQTT

Revpipycontrol.exe from https://revpimodio.org
This web site from Sven Sauger is important / interesting because of the background description of the RevPiPyModio2 environment and all the examples

More info from (YouTube/webinars, et)
MQTT: http://www.steves-internet-guide.com

Pre 2: Before you start with the assignments

The software should be downloaded from the Internet or installed from installed the USB before you switch to the workshop Wi-Fi Smart. This Wi-Fi network is not connected to the Internet, so once on Smart you can't download software any more.

Each Pi with Raspbian has a built-in VNC server. You only need the free VNC viewer on your PC. Install it by downloading from www.realvnc.com (of use the USB stick to install it).

We will use SSH. If you use MS-windows you might use PuTTY from putty.org
(Since April 2018 *ssh* is part of MS-windows 10, but you need to active it using settings, apps and feature, optional features and then OpenSSH. In that case putty is not needed.)

You need on your notebook a Python environment. Skip Python2, go for Python3. Download Python from www.python.org/downloads and select the latest version (*python-3.7.3-amd6.exe* for MS-windows on AMD or Intel processors). You get the IDLE for Python. IDLE is the integrated development environment (IDE) for Python. **Notice that you must select to include Python in your path (bottom menu option in opening installation screen).**

For our assignments we need the revpimodio2 library packages from Sven Sauger.
Open a terminal window with the command prompt and type: *pip3 install revpimodio2*

Background information on Python packages

> Pip is the python installation of packages. In general Python packages come in python source code, interesting once you start debugging.
> In general, the pypi.org is the best place to find python packages and their info.

Finally, download our Python source code demo programs on your computer using GitHub. Go to https://github.com and do a search for ejsol/Smart-Industry. Download the ZIP and extract all.

Finally, this instruction, the instruction for the use of the Pi with the Seeed Grove I/O and the Powerpoint PDF file are on the USB stick too.

Pre 3: Background info on the RevPi and its RevPi Modio2 software environment

The **RevPi is a modern, new type of PLC** based upon standard open hardware (low cost) and open source (powerful) software. It is a shift away from proprietary solutions and can be programmed, still in IEC 61131 SFC PLC languages, but more important in modern computer language as C and Python. In this course we will use Python. Python is easier than C, and be used for logic control, but also for data analysis and visualization and is a common language used in AI programs too.

The **RevPi hardware is based on Raspberry Pi** hardware, although industrialized for more reliability. (In particular the in-the-long-run-not-so-reliable microSD card is replaced by eMMC memory and for cyber security reason it has no Wi-Fi.) There are two version, the Core 1 and Core 3, only use the Core 3 (but this course was developed on a Core 1). The RevPi-Core 3 is a CPU with 4 core unit doing the (parallel) processing. The Core 1 with a CPU with only one core is much slower. There are excellent YouTube instruction videos at the revolution.kunbus.de website on how to start with the RevPi.

For the **Python3 programming for the RevPi** skip the Kunbus environment. Go straight to the revpimodio environment (or actually r**evpimodio2**) of Sven Sauger at www.revpmodio.org . Not only does it include a much easier programming interface, it also includes the RevPiPy environment with the *revpipycontrol.exe* program on Windows (and Mac OS and Debian/Ubuntu/Raspbian Unix). We will start with that environment in the first three assignments. As Revpimodio supports MQTT we will be using this for data communication with data acquisition programs we will be running from our notebook to the RevPi in the subsequent examples

One critical aspect: Old style PLC programming (IEC 61131) was the round-robin control loop of reading the input, processing the code and setting the output, and again and again while guaranteeing a certain max response time (in milli seconds) to react upon an event (changed input). **Modern PLC programming**, thanks to far more powerful CPUs today, **use asynchronous event programming**. Event programming is well known in graphical user interfaces where keyboard and mouse clicks lead to events to be processed. Standard operating systems are supporting event programming very well. As there is no difference between a user interface and I/O events modern PLC are based upon modern computer hardware and programming environments. There exists already a generation conflict between old classical PLC programmer and youngsters trained with IT type programming languages. [1]

[1] The author of this statement was part of the international team that created the original IEC 1131 standard 30 years ago around 1989. Still it is his belief programming industrial controllers will move forward to standard computer programming languages and will be more and more generated from even higher level (formal languages) tools.

Pre 4: Using RevPiPyControl (on your PC) and RevPiPyLoad as service on the RevPi

Remote control of a RevPi from your own computer is done by *revpipycontrol.exe* .[2] That program is also written in Python (with open source code). It allows you to start/stop the RevPi, download programs to the RevPi and to read and write I/O. We first (0) get ourselves acquainted with this program. Later on, we will use another program MQTT.fx (assignment 6) that can read/write the I/O at the RevPi. MQTT.fx allows you to monitor and dive into the MQTT messages and subscribe to the different variables. In this assignment, we will run a Python client program to control RevPi and collect its data. However, as a hint to possible next steps we will end (assignment 7) with an example using node-Red visual programming instead of Python programming.

Before you can use RevPiPyControl, you must start the revpipyload service on the RevPi. That can be done by $ *sudo service revpipyload start* but at the course it is already done automatically during the boot phase of the RevPi (using the systemctl command to include it in the systemd deamon (this is all Unix system mgt talk you can forget for now).)

On your own notebook start the program *revpipycontrol.exe* to control the RevPi as follows:

In the 1st menu option 'Main', create a 'Connection' ('name' it Rev-2, 'IP address' 10.0.0.2 and 'APPLY' it and then 'save' it). Then in 3rd menu option 'Connect', select the Rev-2 and connect. If a connection works, you see a green RUNNING or red NOT RUNNING bar.

Now you can start playing with the I/O's. Select e.g. the 'PCL watch mode' (bottom bar) where you can select '32 | Rev DIO' and then read all the I/O of the RevPi. There you can set the Outputs and then select 'Write Outputs'. See what happens.

3.1 remote login, monitor and control of the RevPi

From your own computer run the program revpipycontrol.exe

Revpipycontrol (from https://revpimodio.org/dnl/revpipycontrol_0.7.1.exe)

main (1st menu), connections, revpi 10.0.0.2 on port 55123, save,
then connect (3rd menu), revpi,

PLC (2nd menu), PLC watch mode, 32|RevPi DIO, read all I/O's
PLC (2nd menu), PL C program, download to your own pc, e.g. desktop
and then open RevPi_event-v1.py file (e.g. with word) and have a look at it.

exit subwindows of PLC download and go to 32|RevPi DIO en select main_relais,
then to main RevPi Python PLC windows, write Outputs (yes). One click, but nothing?
again in main RevPi Python PLC windows, PLC stop and write output again,
and for the fun part select also relais-1, and -2. and write output again.

Finally have a look at the 2nd menu option 'PLC'. If space on the screen, then open the first menu option, 'PLC log'. You will use the 2nd one 'PLC options' to select the Python3 program you will use in the next 3 assignments.

[2] Some virus checkers mark this file as containing a virus. Still unclear if it is serious or a false positive.

Pre 5: Introduction to the demo's/assignments

The demo we use will have three switches (inputs) and three relay (outputs). There is a main_switch and main_relay to turn the system on/off. Once the main switch is turned on, the two other switches can be operated. However, the very simple control logic we will use is that corresponding relay can only turn on, if the corresponding switch is on and the other not.

If main_switch == ON, then main is ON
If main == ON
    If switch 1 == ON and switch 2 == OFF, then relay 1 is ON
    If switch 2 == ON and switch 1 == OFF, then relay 2 is ON

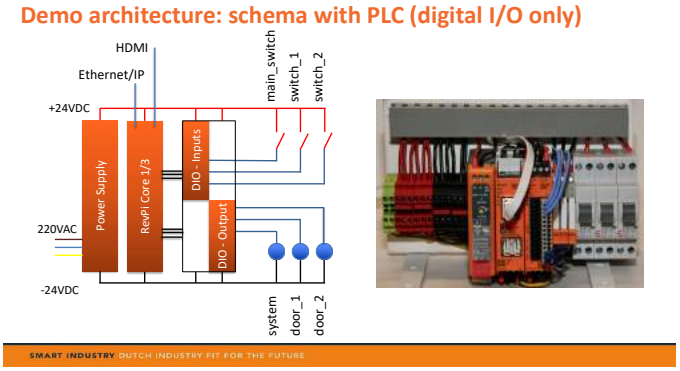
In other words, once the system is turned on, you can only open a door if the other door is locked. The basis logic e.g. for a water lock for passing ships and where you don't want both doors open at the same time.

In our demo we are modelling a climate-controlled warehouse with two doors to minimal false air coming in and out when a door is opened. By having only one door open at a time the mixing of false air with a different temperature and humidity is minimized. In the demo we can count the number of times the doors are opened and how long.

From the received I/O data from the RevPi a Python3 program can monitor the time one of the doors is open and estimate the energy loss due to air mixed between the controlled climate in the warehouse and the air outdoor. In the example is it assumed that once the door is longer open then 10 units, the air is maximally mixed and keeping the door longer open does not have much impact regarding energy loss anymore.

The first three assignment demonstrate the modern way of programming PLC with event driven (of asynchronous) programming, starting with a plain simple main-program version (1), followed by an object-oriented version (2) and finally with a more complete version (3). Although the OO (object oriented) version might initially look like rocket science, it becomes less overwhelming once you start using it. More can be found on the internet and you just start copying and modifying them.

Demo architecture: schema with PLC (digital I/O only)



Basic example program

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

import revpimodio2

def main():
    global main_state, state_1, state_2
    main_state = False
    state_1 = False
    state_2 = False
    rpi = revpimodio2.RevPiModIO(autorefresh=True)

    def event_main_on(ioname, iovalue):
        global main_state
        rpi.io.main_relay.value = True
        main_state = True

    def event_main_off(ioname, iovalue):
        global main_state
        rpi.io.main_relay.value = False
        rpi.io.relay_1.value = False
        rpi.io.relay_2.value = False
        main_state = False

    def event_switch_1_on(ioname, iovalue):
        global main_state, state_1, state_2
        if main_state and not state_2:
            rpi.io.relay_1.value = True
            state_1 = True

    def event_switch_1_off(ioname, iovalue):
        global state_1
        rpi.io.relay_1.value = False
        state_1 = False

    def event_switch_2_on(ioname, iovalue):
        global main_state, state_1, state_2
        if main_state and not state_1:
            rpi.io.relay_2.value = True
            state_2 = True

    def event_switch_2_off(ioname, iovalue):
        global state_2
        rpi.io.relay_2.value = False
        state_2 = False

    # Register event to main_switch, switch_1 and switch_2
    rpi.io.main_switch.reg_event(event_main_on, edge=revpimodio2.RISING)
    rpi.io.main_switch.reg_event(event_main_off, edge=revpimodio2.FALLING)

    rpi.io.switch_1.reg_event(event_switch_1_on, edge=revpimodio2.RISING)
    rpi.io.switch_1.reg_event(event_switch_1_off, edge=revpimodio2.FALLING)

    rpi.io.switch_2.reg_event(event_switch_2_on, edge=revpimodio2.RISING)
    rpi.io.switch_2.reg_event(event_switch_2_off, edge=revpimodio2.FALLING)

    # initialize output to False = 0 = off = no-lights
    rpi.io.main_relay.value = False
    rpi.io.relay_1.value = False
    rpi.io.relay_2.value = False
    rpi.mainloop(blocking=False)

    # Switchthe LED A1 every sec, till rpi.exitsignal.wait returns True after SIGINT/SIGTERM
    while not rpi.exitsignal.wait(1):
        rpi.core.a1green.value = not rpi.core.a1green.value

    #exit/terminate program and reset output to False
    rpi.io.main_relay.value = False
    rpi.io.relay_1.value = False
    rpi.io.relay_2.value = False
    revpimodio2.RevPiModIO(autorefresh=False)

if __name__ == '__main__':
```

1 A simple, most basic Python PLC event program

(Return the RevPi in normal command line mode, stop the windows mode). Use your notebook and start the RevPi Python control program. Select in revpipycontrol (menu 2 'PLC', submenu 2 'PLC options') as Python PLC program name the Python3 program: 1-*Rev- -simple-mainloop-by-revpipycontrol.py*. It starts with importing the revpimodio2 library. Defines the initial states and defines the event procedures. In these event procedures the control logic is programmed. The next part is where the events are registered and linked to changes of the inputs (Rising and Falling). So, if in the main_switch it turned on (rising), the event calls the subroutine event_main_on. In this event the main state become true and the main_relay is turned on, etc. Then at the bottom the program will every second check whether is stopped by the keyboard and turns off the relays and closes the program. Notice the logic of "IF Main == ON AND NOT other door == ON THEN open door" you can find back in the two *event_switch_1 or 2* functions.

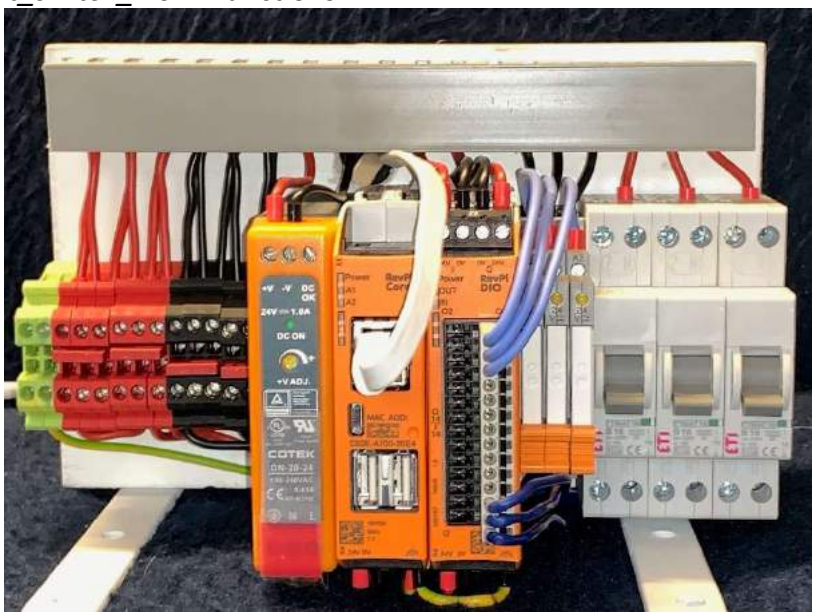


Figure 1 RevPi (core 1) with 24VDC powersupply and DIO connected to 3 input and 3 outputs

2 An object-oriented version of the same program

Select again in revpipycontrol (menu 2 'PLC', submenu 2 'PLC options') as Python PLC program name the Python3 program: 2-*Rev-loop-with-class-by-revpipycontrol.py* This program defines an object-oriented class in which the program is executed. It looks different because of the self.xxxxx. Actually read the self as the rpi.xxxxx. (rpi is the name of the revpi object). A class is the definition of an object with data and subroutines/functions. In general a class has an init() subroutine and the subroutines that get activated when called. Here we do the most in the start subroutine where the mainloop is. Both the program from assignment 2 and this program behave the same.

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

import revpimodio2


class MyRevPiApp:

    def __init__(self):
        """Init MyRevPiApp class."""

        # Instantiate RevPiModIO
        self.rpi = revpimodio2.RevPiModIO(autorefresh=True)

        # Handle SIGINT / SIGTERM to exit program cleanly
        self.rpi.handlesignalend(self.cleanup_revpi)

        # Register event to toggle output O_1 with input I_1
        self.rpi.io.main_switch.reg_event(self.event_main_on, edge=revpimodio2.RISING)
        self.rpi.io.main_switch.reg_event(self.event_main_off, edge=revpimodio2.FALLING)
…..

    def start(self):
        """Start event system and own cyclic loop."""

        # Start event system without blocking here
        self.rpi.mainloop(blocking=False)

        # My own loop to do some work next to the event system. We will stay
        # here till self.rpi.exitsignal.wait returns True after SIGINT/SIGTERM
        while not self.rpi.exitsignal.wait(1):

            # Switch on / off green part of LED A1 to signal to user that PLC runs
            self.rpi.core.a1green.value = not self.rpi.core.a1green.value


if __name__ == "__main__":
    root = MyRevPiApp()
    root.start()
```

3 Example with A1green & A2red light for indication of running system

The final program you can load now is demo 3-Rev-final …. It is practically the same as the program in example 2, but it includes some more lines of code to improve its usability. One usability aspect is blinking of the leds at the RevPi Core module to indicate that an active program is running, A1 green programs runs, A2 red main_state/switch is on. Another one is the handling of ending the program and resetting the outputs in a controlled shut-down state.

```
class MyRevPiApp:

    def __init__(self):
        """Init MyRevPiApp class."""

        # Instantiate RevPiModIO
        self.rpi = revpimodio2.RevPiModIO(autorefresh=True)

        # Handle SIGINT / SIGTERM to exit program cleanly
        self.rpi.handlesignalend(self.cleanup_revpi)

        self.rpi.core.a1green.value = True      # program is loaded and active
        self.rpi.io.main_relay.value = False
        self.rpi.io.relay_1.value = False
        self.rpi.io.relay_2.value = False

        self.main_state = False
        self.state_1_on = False
        self.state_2_on = False

    def cleanup_revpi(self):
        """Cleanup function to leave the RevPi in a defined state."""
        # Switch off LED and outputs before exit program
        self.rpi.core.a1green.value = False
        self.rpi.core.a2red.value = False
        self.rpi.io.main_relay.value = False
        self.rpi.io.relay_1.value = False
        self.rpi.io.relay_2.value = False
        self.rpi = revpimodio2.RevPiModIO(autorefresh=False)

    def event_main_on(self, ioname, iovalue):
        """Called if main_switch goes to True."""
        # Switch on/off output O_1
        self.rpi.core.a2red.value = True
        self.rpi.io.main_relay.value = True
        self.main_state = True

…….
```

4 Bare local RevPi program running from the local Raspbian windows user interface

Normally you would run the RevPi without its own display and no local Raspbian windows user interface. Here we will demonstrate a RevPi with its own screen running Raspbian windows. This mode consumes processing overhead, but thanks to the Raspbian Pi environment, one can easily run with a local keyboard/mouse/display and a Python program with a graphics user interface. The first example is just an introduction with a user interface only to stop the program that controls the I/O using tkinter.py library

Normally you would start the RevPi in command line interface mode (the $...). Now you have to start on the RevPi after booting the windows environment with $ *startx*
And you have to make sure RevPiPyControl is not running, or at least instructing revpipyload to start a RevPi program to run (at startup) as the same time as this program is running. If you do, two programs are controlling the same I/O which will then start to flip-flop at a high frequency.

This program *Rev3-1-control….py* is practically equal to basic example, but it includes the tkinter library *from tkinter import* * and adds a section by extending the __init__ process with declaring the stop button and the event called when the ‘stop running PLC program’ button is clicked-on.

```
Class MyRevPiApp(Frame):

    Def __init__(self, master=None):
    …..
    ….
    # Handle tkinter part
     super().__init__(master)
     self.master.protocol("WM_DELETE_WINDOW", self.cleanup_revpi)

     self.master.wm_title("plc program")
     self.master.wm_resizable(width=False, height=False)
     fontlabel = ('helvetica', 15, 'bold')
     self.btn_stop = Button(self.master, text="Stop running PLC program", font=fontlabel)
     self.btn_stop["command"] = self.cleanup_revpi
     self.btn_stop.grid(row=0, column=0)

def cleanup_revpi(self):
     """Cleanup function to leave the RevPi in a defined state."""
     # Switch off LED and outputs before exit program
     self.rpi.core.a1green.value = False
     self.rpi.io.main_relay.value = False
     self.rpi.io.relay_1.value = False
     self.rpi.io.relay_2.value = False
     self.rpi = revpimodio2.RevPiModIO(autorefresh=False)
     # tkinker
     self.master.destroy()


    ……
```

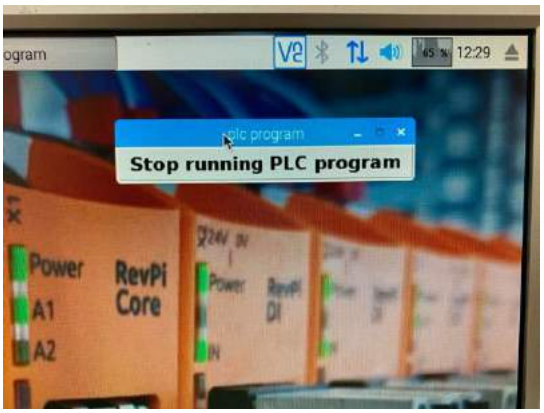


Figure 2 Screen when running Rev3-1-control-only.py

Just start this program from the RevPi windows user interface. Click on it and select execute in terminal mode. This can be done when the RevPi has a display and mouse/keyboard attached. The alternative is that a VNC session has been started from your own notebook to the Rev-2 (10.0.0.2). Notice the little activity icon in the upper right corner at 65% at this core 1. Just before it the program was started running at 100%.

5 Running a RevPi program with a full user interface on the RevPi

Starting with the bare user interface it is possible to create a full dashboard with buttons and sub windows to mimic the hardware switch and relays. The program *Rev3-2-full-UI….py* is however a much longer program because of all the graphics items. Whereas the basic example is only 125 code lines, this version is 250 lines long.

In this program it does not matter whether a push button is clicked or a switch is activated. In both case the control program of turning on/off the relays is performed, but now not only the relay is activated, also the status is updated on the screen.

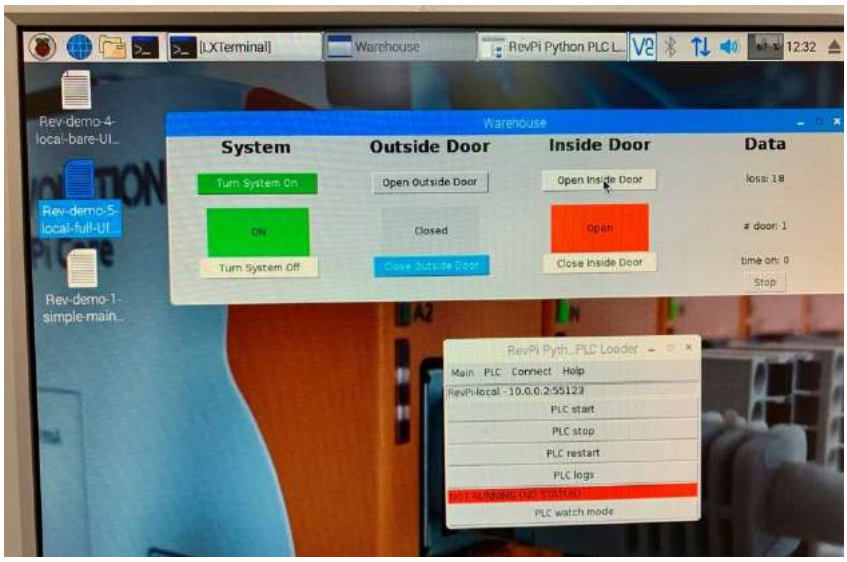


Figure 3 screen when running Rev3-2-full-UI (notice Revpipycontrol is stopped)

In this picture the *revpipycontrol.exe* program is also started in order to show that no other PLC program is running at the same time.

The reason you want to run a local user interface could be for control and informing the user of monitoring data extracted from the process being controlled. Notice on the right side of the windows a data collection, monitoring and analyses function is demonstrated. In this case only one door has been opened and the air/energy loss is at 18 units.

Notice the VNC icon on the topbar, right side. You can also access the RevPi using VNC (on the demo revpi's VNC has been enabled in raspi-configuration program *raspi-config*. Once enabled, you can start a VNC session from your own computer (in the 10.0.0.0/24 netwerk), log in on the RevPi (here 10.0.0.2) with the username and passwords smart/industry and see RevPi screen and control RevPi from remote.

(Of course, enabling VNC involves a security risk, but more critical is the user application where the system can be turned on (and off) from remote. In the coding there are comments on how to remove the turning on of the system main_switch from the user interface, such that the doors can only be controlled from remote once locally the system has been turned on physically by someone physically at the site.)

6 OPC-UA usage and examples (only on a RevPi Core 3)

Note: the RevPi core 1, similar to a Raspberry Pi model 1, is not running OPC-UA properly. It is an older architecture, today you really need a RevPi core 3. If you only have a core 1, MQTT is however running fine (and suitable for smaller applications).

Open Platform Communication – Unified Architecture, also known as IEC 62541, the defector standard for equipment communication within Smart Industry (NL) or the Industrie 4.0/4IR community.

We use the open source python version FreeOpcUa from Olivier Roulet-Dubonnet. (from github.com). There is the *Rev3-4-server-OPC.py* that should run on the RevPi (core3, in our case below the 10.0.0.3, but in the demo setup in 192.168.0.0/24). Once started you can start a client program on another computer in the same network, e.g. Rev3-4-…-client-OPC.py This program collects data to be stored in the data base. From the github.com or pypi.org you can also download the opcua-client-gui, install it and the connect to 10.0.0.3:4840, inspect the object model with the variables and monitor events and construct a graph. (Installing the client on a MS-windows pc requires the very extensive winpython.github.com program 600MB and a slow installation).

This is a section in the server program where it is started.

```
print('starting OPC server . (url 10.0.0.3:4840)')
self.opc_server = Server(shelffile="/home/pi/RevPi-OPC-Server")
self.opc_url = "opc.tcp://10.0.0.3:4840"
self.opc_server.set_endpoint(self.opc_url)
# TODO security
print('starting OPC server .. (namespace)')
self.opc_name = "RevPi-opcua-server"
self.addspace = self.opc_server.register_namespace(self.opc_name)
print('starting OPC server ... (variables)')
self.opc_node = self.opc_server.get_objects_node()
self.param = self.opc_node.add_object(self.addspace, "Parameters")

self.opc_time = self.param.add_variable(self.addspace, "Time", 0)                          # opc i=2
self.opc_trigger = self.param.add_variable(self.addspace, "Trigger", 0)                    # opc i=3
self.opc_warehouse_state = self.param.add_variable(self.addspace, "System state", 0)       # opc i=4
self.opc_door_outside = self.param.add_variable(self.addspace, "Outside door", 0)          # opc i=5
self.opc_door_inside = self.param.add_variable(self.addspace, "Inside door", 0)            # opc i=6
self.opc_open_percentage = self.param.add_variable(self.addspace, "Door open %", 0.0)  # opc i=7
self.opc_door_outside_share_percentage = self.param.add_variable(self.addspace, "Outside/Inside share", 0.0)

self.opc_time.set_writable()
self.opc_trigger.set_writable()
self.opc_warehouse_state.set_writable()
self.opc_door_outside.set_writable()
```

The ‘shelffile’ is a trick with FreeOpcUa to speed to loading time in subsequent starts of the server. The opc_... variables are stored in the object node and can later be accesses in a client program with notions as e.g. ns=2;i=2 . To see which opc_variables a server has, the client will read the node info after a client connect as e.g. in the piece of python code:

```
while True:
    try:
        time_stamp = client.get_node("ns=2;i=2")
        trigger = client.get_node("ns=2;i=3")
        warehouse_state = client.get_node("ns=2;i=4")
        door_outside = client.get_node("ns=2;i=5")
```

On the next page you see a screen dump of the FreeOpcUa client gui where you can see the opc-variables and the browse name, the subscription monitoring certain variables and, in this case, the graph with the percentage of open-doors while the system is on (Doors open %) and the share between the outside and inside door. In the running Rev-demo-7-client… program below you see the variables, as doors open % is e.g. in this case 16-17%

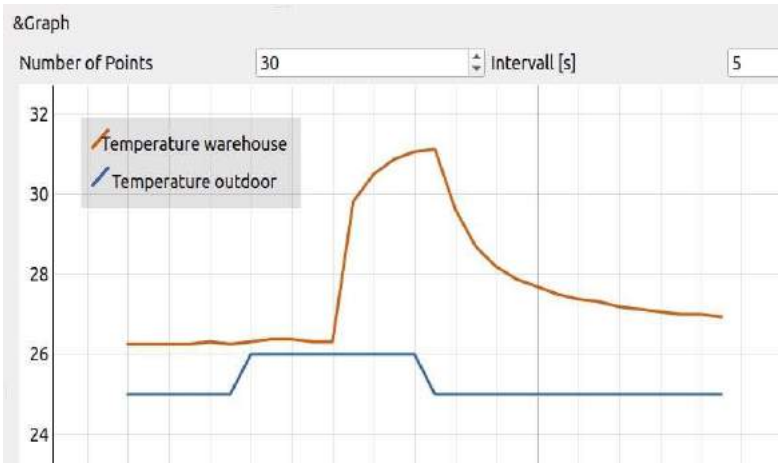
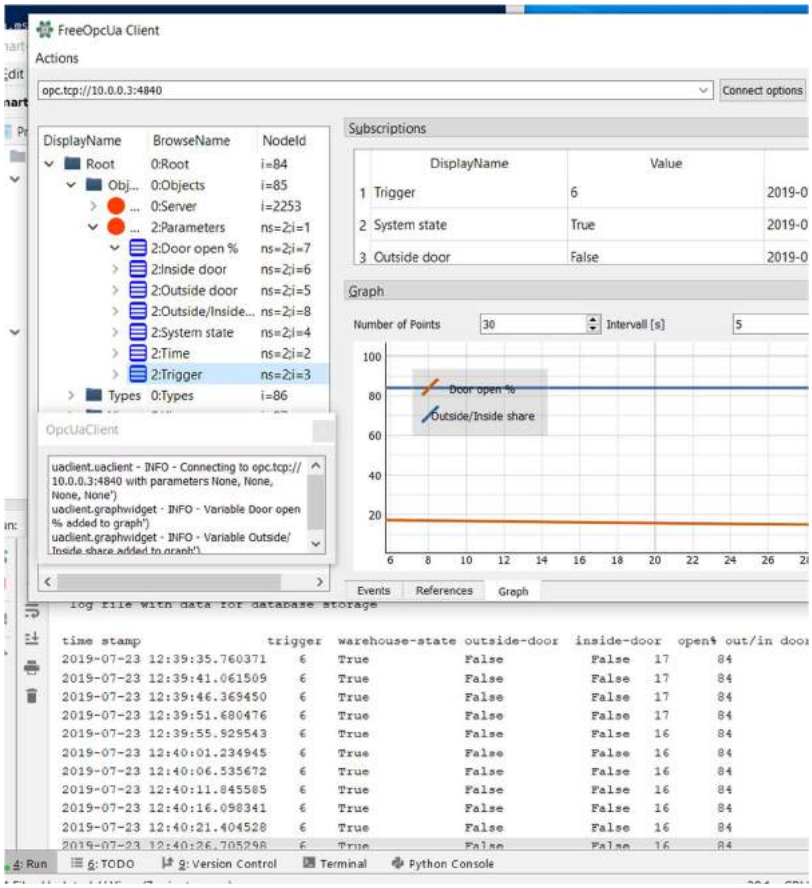


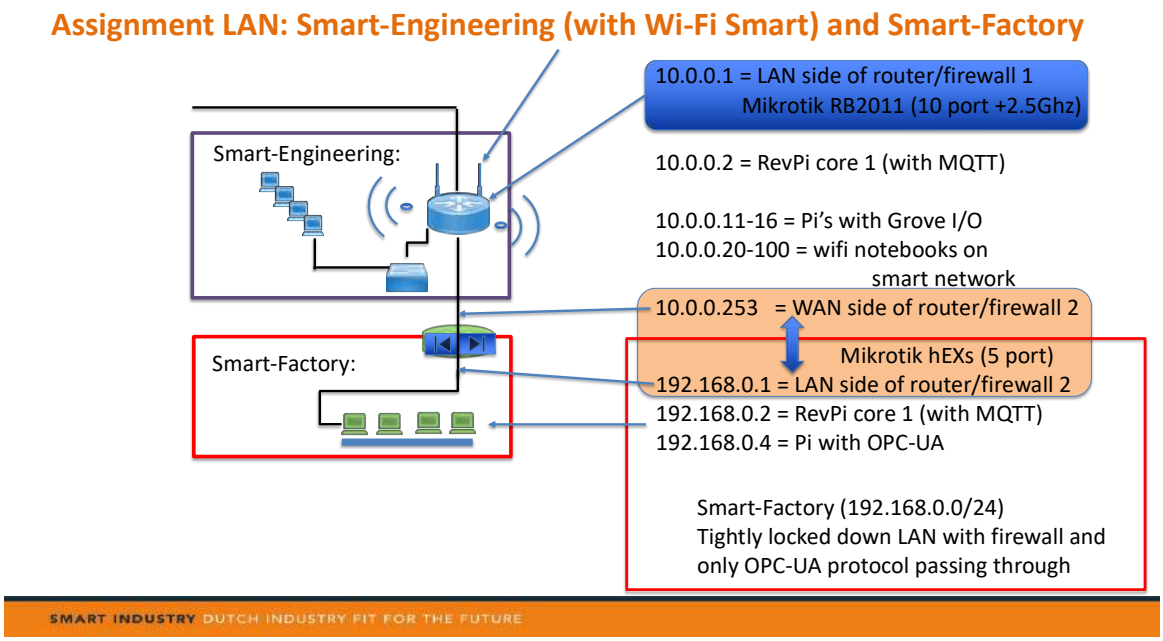
Figure 4 Output from Opcua-client using Rev3-4-server



7 OPC-UA server behind firewall, OPC-UA client outside firewalled network

The OPC-UA server will often run on a machine in a highly secured own network segment, after a tightly locked down firewall. The client could be running outside this network. E.g. the Pi with the Groove I/O and the OPC-UA server run in their own e.g. 192.168.0.0/24 network wired to the router/firewall on the LAN site and on the WAN site the router/firewall is in a e.g. 10.0.0.0/24 office network (with Wi-Fi to notebooks). Now the client cannot access the OPC-UA server.

In this assignment no Python program is written, but we need to configure the firewall and add specific parameters in the Python client program. From the client side the server is not visible, only the WAN port of the firewall e.g. 10.0.0.253. And if you have multiple OPC-UA servers (more Pi's e.g. 192.168.0.4 & 192.168.0.11) how to access them if you only have one access point?



The common way of working is to use so-called Destination NAT (network address translation). Inside the LAN part Pi-4 and Pi-11 and a client can use the IP addresses and the OPC-UA port of 4840. Now the firewall-LAN side, e.g. 192.168.0.1 can also act as client to the two Pi's. In the firewall-LAN DstNAT is now configured such that is a message from extern is received on IP nr (firewall) 10.0.0.253:48404 (or any other portnumber above 1024 and below 64K, but preferable between 50K-64K)) is mapped to 192.168.0.4:4840 and similar to port 48411 is mapped to 192.168.0.11:4840. This way a request from the client is translated by the firewall and forwarded to the proper Pi and during the answer from the Pi, the firewall knows to with client the answer should be sent.

This is a common way also used with webservers behind firewall. The only thing to do is to configure the firewall to accept DstNAT and enter the translations in the firewall rules set of the firewall. (see training slides for more details too). SrcNAT is disabled, so not access to the internet is possible.

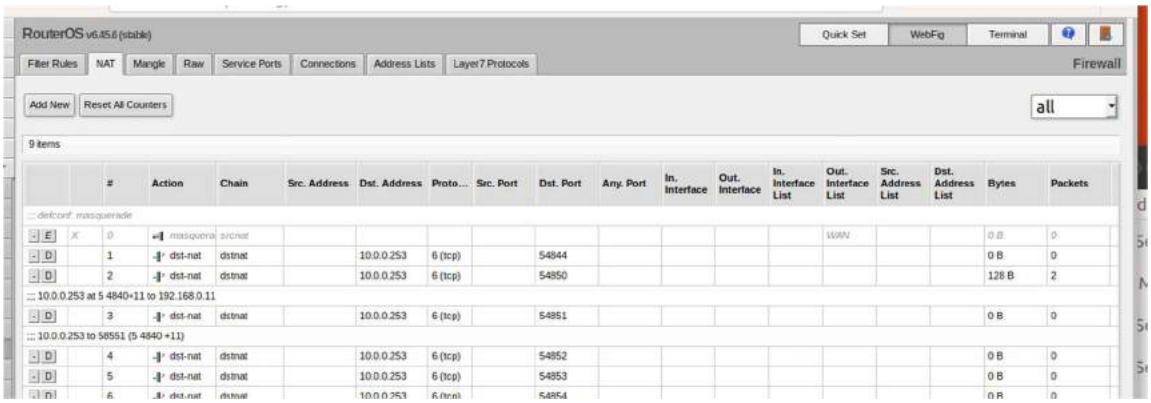


Figure 5 NAT firewall dstnat rules (notice scrnat is disabled)

/ip firewall filter
chain input (to router itself)
add action=drop chain=input comment="drop invalid to firewall router at 192.168.0.1/24" connection-state=invalid
add action=accept chain=input comment="allow established connections to firewall router " connection-state=established

allow connection to firewall router from local network (!ether1 implies ether2-5 (=LAN) as ether1 is WAN),
so next rules state accept all local LAN traffic, drop all remain WAN traffic
add action=accept chain=input comment="" in-interface=!ether1 src-address=192.168.0.0/24
add action=drop chain=input comment="drop all to firewall router not coming from LAN" in-interface=ether1

chain forward from WAN (ether port 1) to LAN (ether port 2-5) or vice versa
add action=drop chain=forward comment="drop invalid" connection-state=invalid
add action=accept chain=forward comment="accept established and related" connection-state=established,related
add action=drop chain=forward comment="drop all from WAN not DSTNATed" connection-nat-state=!dstnat connection-state=new in-interface-list=WAN
add action=drop chain=forward comment="drop everything else " disabled=yes

/ip firewall nat
scrnat (source network address translation is e.g. web request (port 80) from a PC to external webserver,
dstnat is request from outside via firewall to internal device, here OPC-server
add action=masquerade chain=srcnat comment="masquerade" disabled=yes ipsec-policy=out,none out-interface-list=WAN
add action=dst-nat chain=dstnat dst-address=10.0.0.253 dst-port=48403 log=yes protocol=tcp to-addresses=192.168.0.3 to-ports=4840

in LAN everyone can call OPC server at ocp:tcp://192.168.0.3:4840 (port 4840).
From outside call the OPC server via router 10.0.0.253 at port 48403

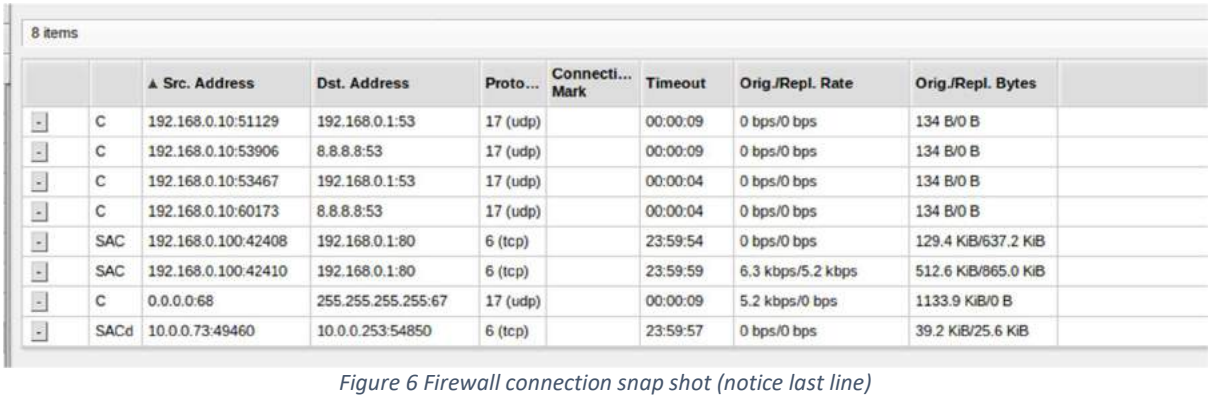


Figure 6 Firewall connection snap shot (notice last line)

P.s with more OPC-UA servers inside the LAN, more dst-nat rules are to be entered per server with other port numbers.