TECHNICAL DOCUMENTATION
IOT GATEWAY

By:

Nicholas Putra Rihandoko **ITB de Labo**

Nauval Chantika **ITB de Labo**

NAKAYAMA IRON WORKS
2023

Instructions

This documentation will explain about the workflow and how to modify of 5 different programs:

1. Simhat Setup Scripts
2. MODBUS Communiction Scripts
3. CANBUS Communication Scripts
4. Authenticator Scripts
5. HMI Scripts

Those programs are used for NePower Project for defaults, so please make some modification before install the program depends on your needs. You can check how to install in here:

IoT Gateway-Installation document

List of Contents

List of Figures

List of Tables

Chapter 1. SIM Hat System (IoT Gateway System)

1.1. Script Installation Standalone

After the SIM Hat module (hardware) is installed, power up the RaspberryPi and copy the folder `***simhat_code***` to Home Folder directory in `***/home/<user>/NIW***` or `***~/NIW***` as shown in **Figure 2**. Also, make sure that you have already setup a ZeroTier network. You can check ZeroTier instructions here. Next, open terminal and run `***init_dial.bash***` by executing the command in **Figure 3**. Continue by following the instructions on the terminal.

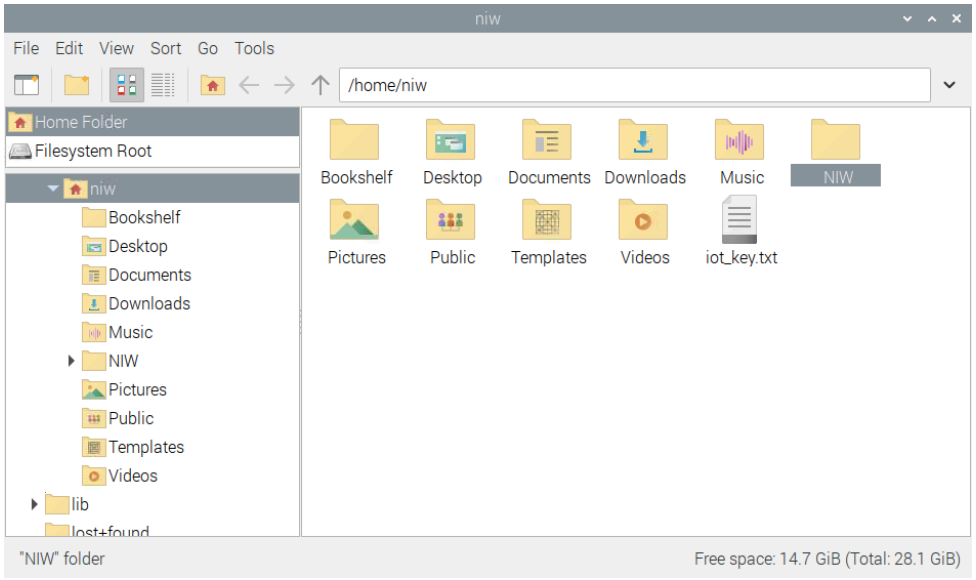

Figure 1 Main Directory for all Nakayama Programs

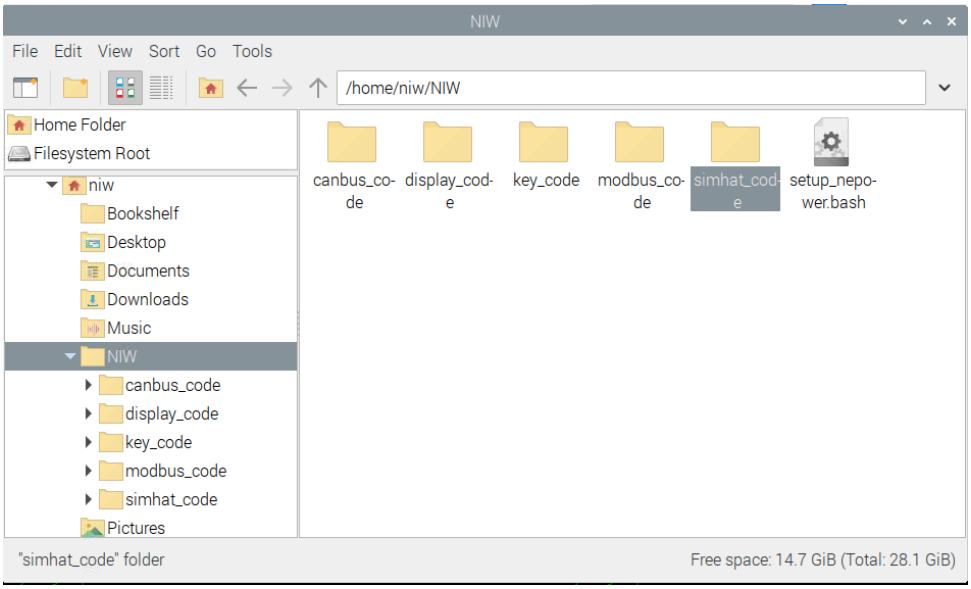

Figure 2 SIM Hat programs Directory

```
sudo bash NIW/simhat_code/init_dial.bash
```

Figure 3 Bash command to install the IoT system standalone

When it asks the question as shown in **Figure 4**, press '***y***' to add router configuration in the installation procedure or press '***n***' to only install SIM Hat for RaspberryPi's internet access only. If you select '***n***', fill in the SIM card's information (APN, username, and password) as well as network ID of ZeroTier network information (check on ZeroTier Central) according to the instructions on the terminal. Wait for the SIM Hat system installation to finish with the message as shown in **Figure 5**. Now, you can access the Raspberry Pi remotely through ZeroTier network.

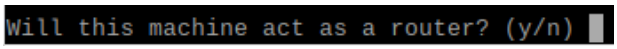

Figure 4 Choice of SIM Hat system configurations

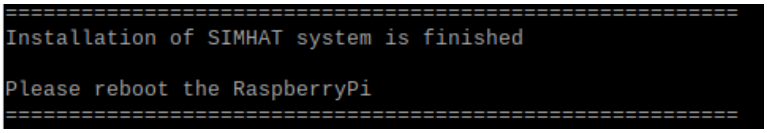

Figure 5 Message indicating SIM Hat system is installed

If you select '***y***', you can choose to skip the SIM Card configuration or not. You also need to determine the subnet of managed local network (Ethernet) address that you want for the router functionality (IoT gateway). The script will automatically install the necessary package to enable routing functionality. The installation is complete when you get the message on your terminal as shown in **Figure 6**.

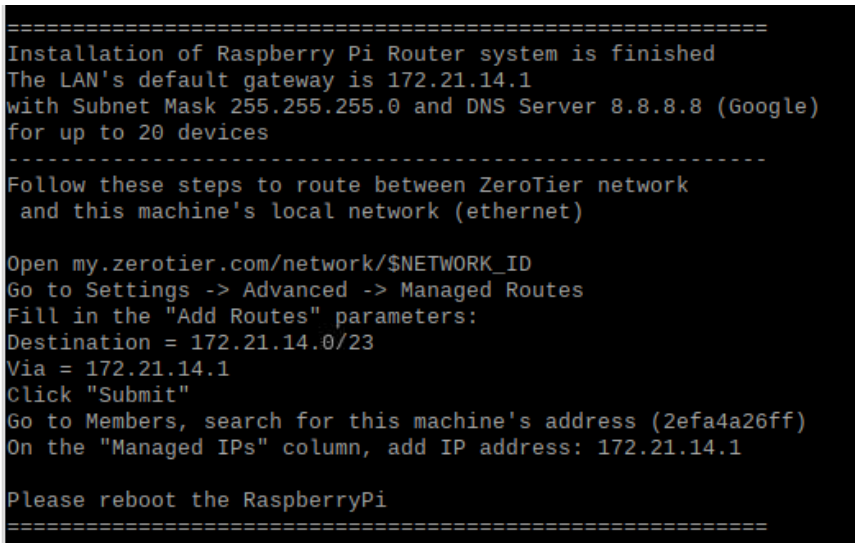

Figure 6 Message indicating IoT Gateway system is installed

In this example, we want to use subnet 172.21.14.0/24 wih 172.21.14.1 as default gateway. Following the instructions on **Figure 6**, login to [ZeroTier Central] then go to the network that the RaspberryPi is connected to. In order to setup remote access from ZeroTier devices to RaspberryPi's managed local network (Ethernet), add the local network routes and click submit as shown in **Figure 7**. Lastly, add the same IP address to the RaspberryPi as shown in **Figure 8**.

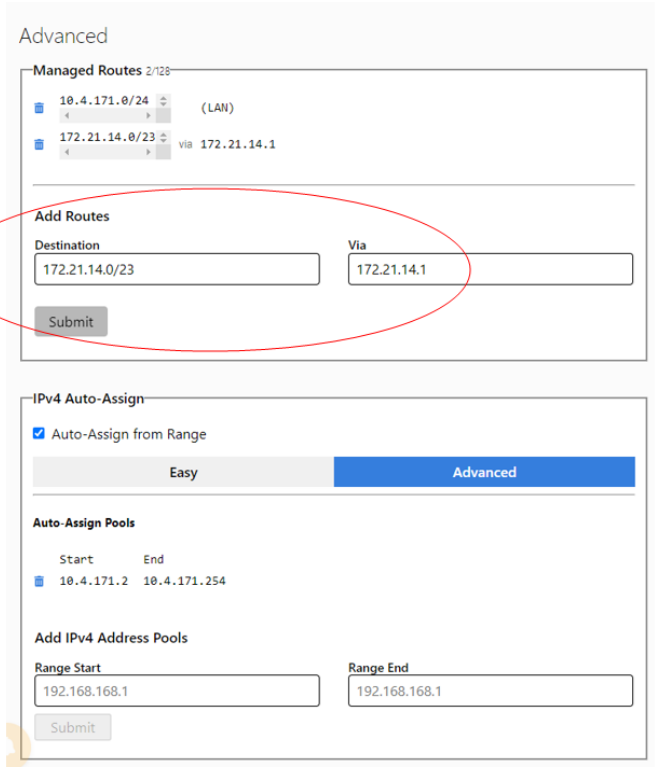

Figure 7 Adding routes in ZeroTier

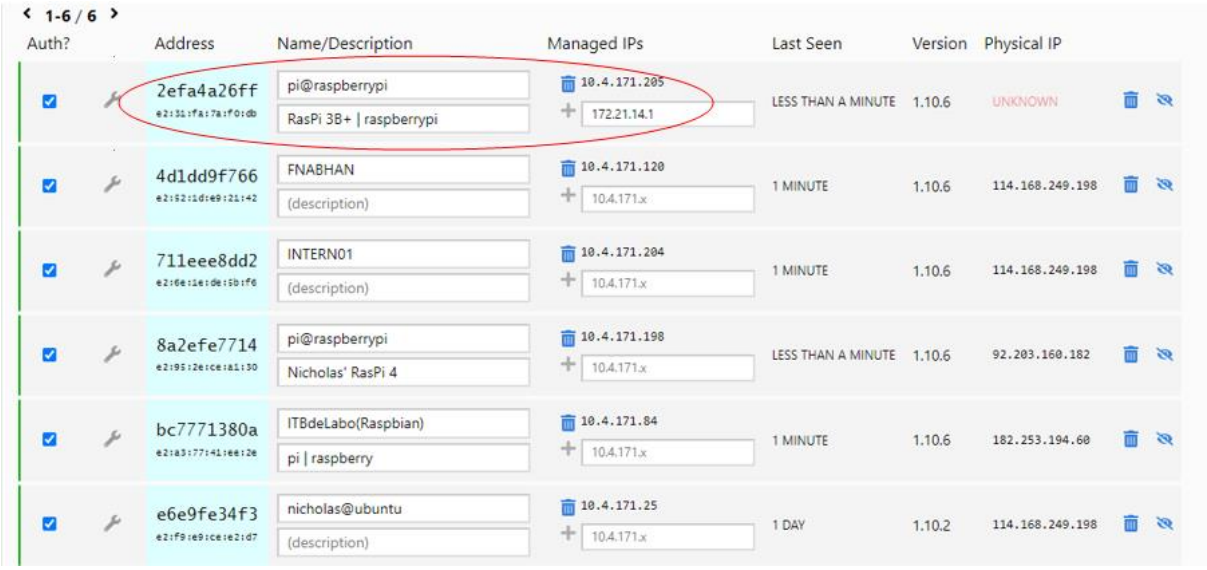

Figure 8 Adding IP address in ZeroTier

1.2. System Architecture

The foundation for the IoT Gateway is based on ZeroTier Virtual Local Area Network (LAN) system as shown in **Figure 9**. Device that is registered in the ZeroTier network can communicate with other devices in the same network as if they were in the same LAN as long as they are online (connected to the internet). If you configure the RaspberryPi as router (IoT Gateway), then you can also communicate with other devices on that RaspberryPi's LAN (connected by Ethernet cable) as long as the RaspberryPi is online.

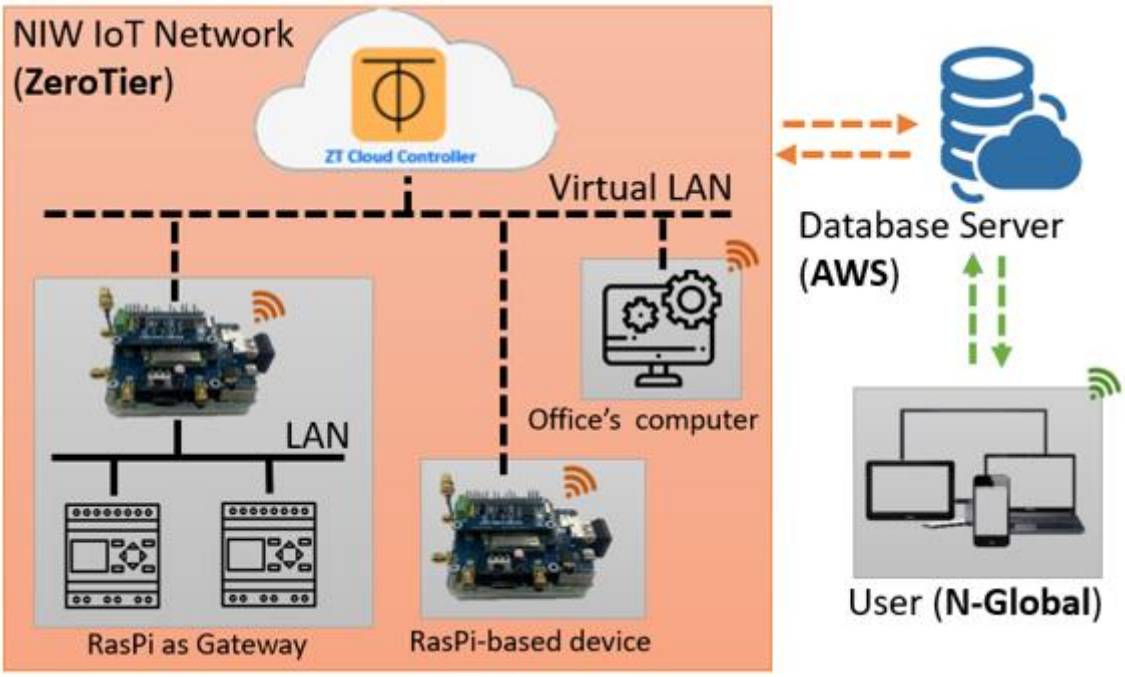

Figure 9 IoT Gateway System Architecture

To access the device that is connected to RaspberryPi's Ethernet, you can do IP scan from you're the office's computer. Based on example in Figure 1.7, the subnet address of the RaspberryPi's Ethernet that needs to be scanned will be '172.21.14.0/24' or '172.21.14.1-254'. You can use Advanced IP Scanner to do this as given example in **Figure 10**. Remember that the the office's computer that is used for remote access must also join the same ZeroTier network as the RaspberryPi. See ZeroTier Knowledge Base for further information and FAQ.

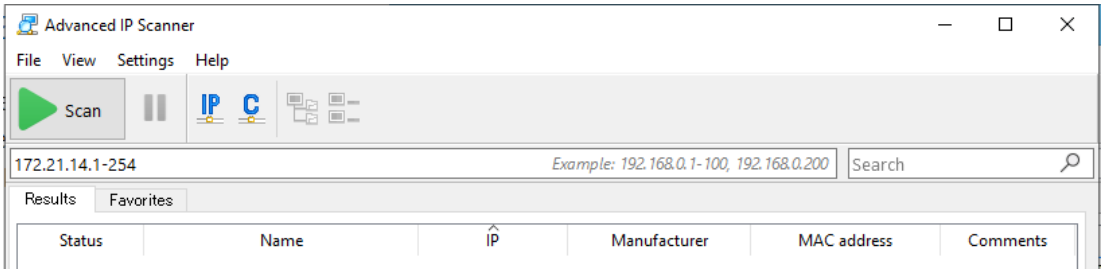

Figure 10 Advanced IP Scanner to obtain IP adresss of RaspberryPi's LAN

1.3. Troubleshooting

If there are any trouble during installation or a need to modify the system, follow the flowchart in **Figure 11** to understand the installation script's logic. Alteration may be applied appropriately to meet specific requirements.**Table 1** maps the corresponding steps in **Figure 11** to the scripts in the folder `***simhat_code***`.

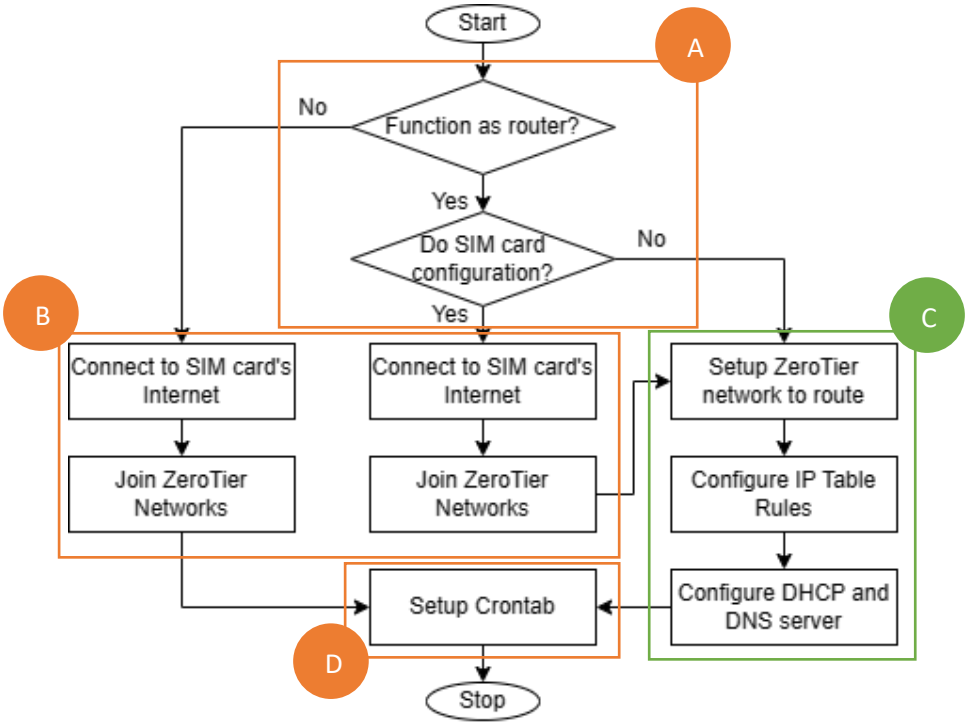

Figure 11 Flowchart of 'simhat_code'

Table 1 Relevant Scripts based on the Flowchart of *'simhat_code'*

Part	File's name	Line	Notes
A	alter_dial.bash	34-62	*$yn_router* and *$yn_sim* is equal to the yes/no question in the flowchart
B	alter_dial.bash	66-187	The script allows one machine to connect to multiple ZeroTier networks
C	route.bash	all	Modify this script to configure routing (gateway) functionality
	alter_dial.bash	275-355	
D	alter_dial.bash	357-378	-

Chapter 2. Modbus System

2.1. Script Installation Standalone

After the RS485/CAN Hat module (hardware) is installed, power up the RaspberryPi and copy the folder `***modbus_code***` to Home Folder directory in `***/home/<user>/NIW***` or `***~/NIW***` as shown in **Figure 12**. Next, open terminal and run `***init_modbus.bash***` by executing the command in **Figure 13**. Continue by following the instructions on the terminal.

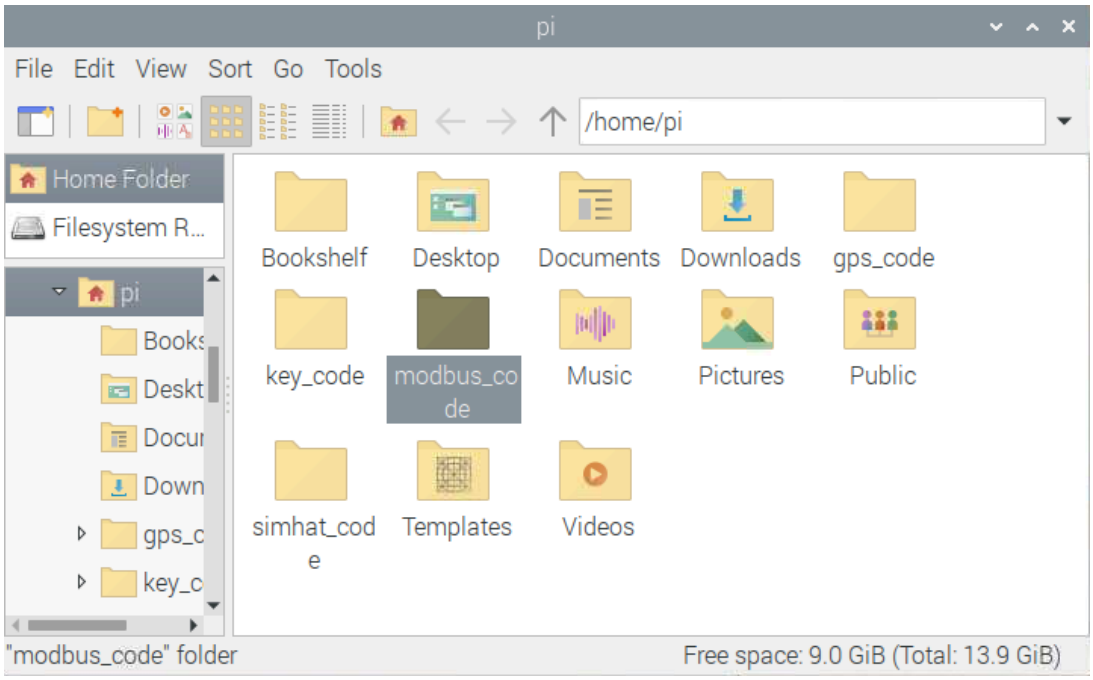

Figure 12 Modbus system installation script directory

```
sudo bash NIW/modbus_code/init_modbus.bash
```

Figure 13 Bash command to install the Modbus system

The script will automatically install the necessary package to enable the communication through RS485/CAN Hat module. The installation is complete when you get the message on your terminal as shown in **Figure 14**.

```
============================================================

Installation of Modbus system is finished

Please reboot the RaspberryPi, just in case :)

============================================================
```

Figure 14 Message indicating Modbus system is installed

2.2. Program Architecture

The code is written in Object Oriented Programming (OOP) on Python environment with PyModbus library as its foundation. The main program is a script called `***main_modbus.py***` inside the `***modbus_code***` folder, while the device-dependent scripts are located in `***modbus_code/lib***` folder. There is also a script called `***query.py***` that is meant for writting/defining specific functions that will be used in the most main programs, not limited to Modbus system only. More about `***query.py***` will be explained in **Chapter 4**. **Figure 15** shows the example of Modbus system's directory.

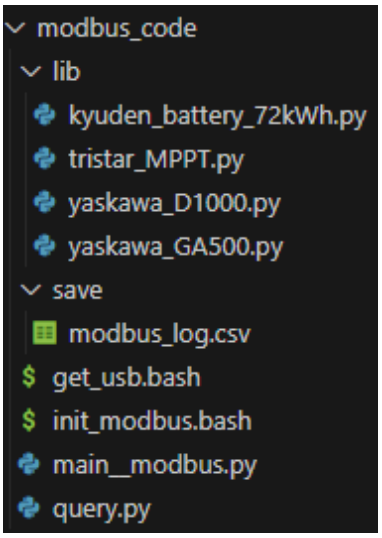

Figure 15 Example of Modbus system's directory

For any projects using the Modbus system, simply create a copy of the `***modbus_code***` folder, replace the scripts inside `***modbus_code/lib***` following the Modbus devices used, then modify the `***main_modbus.py***` as necessary. **Figure 16** show the general flowchart of `***main_modbus.py***`.

The `***setup_modbus()***` function is the mandatory function which initialize the Modbus communication and the device-specific libraries. An example of this function, including its input arguments are shown in **Figure 17**. When there are more than one communication interface (for example in this case is RS-485 and RS-232c), then the program will act as a client for each network interface. Then, each device is initialized as an object using appropriate device-specific library and parameters. To ease working with many devices/slaves, all devices are compiled into one array called `***server***`.

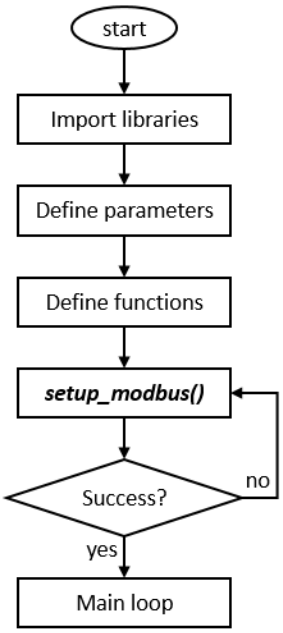

Figure 16 Flowchart of `***main_modbus.py***`

Once initialized, the main program (`***main_modbus.py***`) can send read or write command for each device/slave and it will automatically utilize PyModbus library to communicate with the actual device/slave. For "read" command, the device/slave object will calculate and generate a new attribute according to the information that you want (including the dependencies). There are two ways to send the "read" command. First is by passing an array argument filled with list of address to be read (integer; hexadecimal or decimal). If the address is not yet available in `***self._memory_dict***`, then the generated attribute's name will be exactly the address in hexadecimal (Ex: '0x00A2'). The second method is by passing an array argument filled with attribute name to be calculated. If the corresponding attribute has dependencies, the program will try to obtain the dependencies as well. The attribute name in this method is case-insensitive. See **Table 2** for more.

Table 2 Available Methods in the Modbus Device-specific Library

Purpose	Method Syntax
Read Modbus addresses (*integer*) or calculate custom values (*string*)	obj.send_command(command="read", address=[*integer,integer,…*], fc=*integer**) obj.send_command(command="read", address=[*string,string,…*], fc=*integer**) obj.send_command(command="read", address=[*integer,string,…*], fc=*integer**)
Write parameter on Modbus memory address (*integer*) or do predetermined action (*string*)	obj.send_command(command="write", address=*integer*, param=*integer**, fc=*integer**) obj.send_command(command="write", address=*string*, param=*integer**)
Obtain the attribute name ($string_n$) and value ($value_n$) of the object using its Modbus address ($integer_n$)	[[$string_1$,$value_1$], [$string_2$,$value_2$],…] = obj.map_read_attr([$integer_1$, $integer_2$,…])
Reset all attribute values from "read" command to zero	obj.reset_read_attr()

* may be omitted if the information is available in `***self._memory_dict***`of the device-specific library

```python
# Import library
import datetime # RTC Real Time Clock
import time
import os
import socket
import threading
import logging
from pymodbus.client import ModbusSerialClient as ModbusClient
import query
from lib import kyuden_battery_72kWh as battery
from lib import yaskawa_D1000 as converter
from lib import yaskawa_GA500 as inverter
#from lib import tristar_MPPT as charger

# Logging setup
logging.basicConfig(level=logging.INFO, format="%(asctime)s - %(levelname)s - %(message)s")

# Define Modbus communication parameters
port            = '/dev/ttyS0'    # for RS485/CAN Hat
port_id0        = 'Prolific_Technology_Inc' # for USB-to-RS232C adaptor
port0           = os.popen('sudo bash {}/get_usb.bash {}'.format(os.path.dirname(os.path.abspath(__file__)), port_id0)).read().strip()
method          = 'rtu'
bytesize        = 8
stopbits        = 1
parity          = 'N'
baudrate        = 9600   # data/byte transmission speed (in bytes per second)
client_latency  = 100   # the delay time master/client takes from receiving response to sending a new command/request (in milliseconds)
timeout         = 1   # the maximum time the master/client will wait for response from slave/server (in seconds)
interval        = 30   # the period between each subsequent communication routine/loop (in seconds)

def setup_modbus():
    global port, port0, method, bytesize, stopbits, parity, baudrate, client_latency, timeout
    # Set each Modbus communication port specification
    client = ModbusClient(port=port, method=method, stopbits=stopbits, bytesize=bytesize, parity=parity, baudrate=baudrate, timeout=timeout)
    client0 = ModbusClient(port=port0, method=method, stopbits=stopbits, bytesize=bytesize, parity=parity, baudrate=baudrate, timeout=timeout)
    # Connect to the Modbus serial
    client.connect()
    client0.connect()
    # Define the Modbus slave/server (nodes) objects
    bat = battery.node(slave=1, name='BATTERY', client=client0, delay=client_latency, max_count=20, increment=1, shift=0)
    conv = converter.node(slave=2, name='CONVERTER', client=client, delay=client_latency, max_count=20, increment=1, shift=0)
    inv = inverter.node(slave=3, name='INVERTER', client=client, delay=client_latency, max_count=20, increment=1, shift=0)
    #chr = charger.node(slave=4, name='SOLAR CHARGER', client=client, delay=client_latency)
    server = [conv, bat, inv]
    return server
```

Figure 17 Example of `***setup_modbus()***` function

For write command, there are also two available methods, either by passing an address argument of only single integer value (decimal or hexadecimal), or a string which corresponds to an attribute in the device-specific library. In addition, the parameter that is to be written (integer; hexadecimal or decimal) needs to be passed in the *param* argument. The *param* argument can be a single value, or multiple values in an array. In the multiple values case, the parameters will be written to the adjacent address (increment up), with the address in the ***send_command*** input argument being the first address. The full list of available method can be seen in **Table 2**.

2.3. Device-Specific Library

As shown in **Figure 17**, each Modbus device (slave/server) is an object which is created using a device-specific library which is a Python *class* called `***node***`. At initializations (***__init__***), it created several object attributes (variable/parameter) which are used exclusively in the class's methods (function). These attributes can be categorized generally into 4 categories as explained in **Table 3**. If the information you want is not listed in the `***self._memory_dict***`, you can edit the library by adding a

nested dictionary DataType into the `***self._memory_dict***`. The same can be done for the `***self._extra_calc***`. An example of the dictionary's *key:value* syntax is given in **Figure 18** and is explained in **Table 4** and **Table 5**.

Table 3 Private Attributes in `***node***`Class of Modbus Device-Specific Library

Category	Attribute	Purpose
To ease programming	***self._name***	Give individual name to object created using this library
Parameters for PyModbus communication	***self._slave***	Unit number (ID) of the device/slave in the communication line
	self._client	Serial port to be used by PyModbus
	self._client_transmission_delay	Delay for each subsequent Modbus command/request (in seconds)
Parameters for Modbus read address arrangement	***self._max_count***	Limit the maximum address to be read/write in one command
	self._shift	Handle problematic devices with shifted addresses
	self._inc	Handle increments between each valid address
Related to device's Modbus address & data calculation	***self._memory_dict***	Information related to Modbus device's built-in memory address and data calculation
	self._extra_calc	Information related to custom value that is calculated from Modbus device's built-in data

Table 4 *key:value* Syntax of `***self._memory_dict***`

SYNTAX	"attr_name":{"fcr":fcr,"fcw":fcw,"address":address,"scale":scale,"bias":bias,"round":round,"param":param }	
Key	**Value (DataType)**	**Notes**
attr_name	*dictionary*	The calculated data will be saved into the attribute named ***self.attr_name***
fcr	*integer*	Default function code, can be in hexadecimal or decimal; "read" command: 0x03 (3), 0x04 (4), or None
fcw	*integer*	Default function code, can be in hexadecimal or decimal; "write" command: 0x06 (6), 0x10 (16), or None
address	*integer*	Modbus memory address for this particular data (hexadecimal or decimal)
scale	*float* or *integer*	Used to calculate the data from raw data: $data_{\ calculated} = (scale \times data_{\ raw}) + bias$
bias	*float* or *integer*	
round	*integer*	The number of digits behind decimals of calculated data (rounded)
param	*integer*	Default parameter for write command, can be in hexadecimal or decimal; may be left out if not needed

Table 5 *key:value* Syntax of `***self._extra_calc***`

SYNTAX	"attr_name":{"scale":scale,"bias":bias,"round":round,"limit":limit, "scale_dep":scale_dep,"bias-dep":bias_dep,"compile":compile}	
Key	**Value (DataType)**	**Notes**
attr_name	*dictionary*	The calculated data will be saved into the attribute named ***self.attr_name***
scale	*float/integer*	Used to calculate the data from raw data: $data_{\ calculated} = (scale \times data_{\ raw}) + bias$
bias	*float/integer*	
round	*integer*	The number of digits behind decimals of calculated data (rounded)
limit	*[float,float]*	Used to round the calculated data for specific threshold; Example, with ***"limit":[x,y]*** we get: $data_{\ rounded} \begin{cases} data_{\ rounded} = y \mid data_{\ calculated} < x \\ data_{\ rounded} = data_{\ calculated} \mid data_{\ calculated} \geq x \end{cases}$ Leave empty if not needed (Set value to be **[ ]**)

scale_dep	*[[float/integer,string],…]*	Dependencies* (*string*) paired with a constant (*float*) which are used for calculating raw data; Example, with ***"scale_dep":[[a,"dep_a"], [b,"dep_b"]]*** and ***"bias_dep":[[c,"dep_c"], [d,"dep_d"]]*** we get: $data_{raw} = (dep_a{}^{a} \times dep_b{}^{b}) + (c \times dep_c) + (d \times dep_d)$ Leave empty if not needed (Set value to be **[ ]**)
bias_dep	*[[float/integer,string],…]*	
compile	*[[string,string,…], [string,string,…],…]*	Several attributes* (string) that are compiled into a single array or nested array; may be left out if not needed

* The dependencies to be calculated and attribute to be compiled can be both from `***self._memory_dict***`or `***self._extra_calc***` as long as their *key:value* pairs are written before/above the *key:value* pair that use it

```
# Commands and memory address that are available/configured, add if needed
self._memory_dict = {
    ## read scaling values
    "V_PU_hi":              {"fcr":0x03, "fcw":None, "address":0x0000, "scale":1, "bias":0, "round":0},
    "V_PU_lo":              {"fcr":0x03, "fcw":None, "address":0x0001, "scale":1/(2**16), "bias":0, "round":5},
    "I_PU_hi":              {"fcr":0x03, "fcw":None, "address":0x0002, "scale":1, "bias":0, "round":0},
    "I_PU_lo":              {"fcr":0x03, "fcw":None, "address":0x0003, "scale":1/(2**16), "bias":0, "round":5},
    ## read
    "Battery_Voltage":      {"fcr":0x03, "fcw":None, "address":0x0018, "scale":1, "bias":0, "round":0},
    "Array_Voltage":        {"fcr":0x03, "fcw":None, "address":0x001B, "scale":1, "bias":0, "round":0},
    "Battery_Current":      {"fcr":0x03, "fcw":None, "address":0x001C, "scale":1, "bias":0, "round":0},
    "Array_Current":        {"fcr":0x03, "fcw":None, "address":0x001D, "scale":1, "bias":0, "round":0},
    "Heatsink_Temperature": {"fcr":0x03, "fcw":None, "address":0x0023, "scale":1, "bias":0, "round":0},
    "Battery_Temperature":  {"fcr":0x03, "fcw":None, "address":0x0025, "scale":1, "bias":0, "round":0},
    "Output_Power":         {"fcr":0x03, "fcw":None, "address":0x003A, "scale":1, "bias":0, "round":0}, # Watt
    "Input_Power":          {"fcr":0x03, "fcw":None, "address":0x003B, "scale":1, "bias":0, "round":0}, # Watt
    "Total_Wh_Charge_Daily":{"fcr":0x03, "fcw":None, "address":0x0044, "scale":1, "bias":0, "round":0}
    }
# Used to shift the Modbus memory address for some devices
for key in self._memory_dict: self._memory_dict[key]["address"] += shift
# Extra calculation for parameters/data that is not readily available from Modbus, add if needed
self._extra_calc = {
    "V_PU":                 {"scale":1, "bias":0, "round":5, "limit":[], "scale_dep":[[1,"V_PU_hi"]], "bias_dep":[[1,"V_PU_lo"]]},
    "I_PU":                 {"scale":1, "bias":0, "round":5, "limit":[], "scale_dep":[[1,"I_PU_hi"]], "bias_dep":[[1,"I_PU_lo"]]},
    "Battery_Voltage":      {"scale":1/(2**15), "bias":0, "round":2, "limit":[], "scale_dep":[[1,"Battery_Voltage"],[1,"V_PU"]], "bias_dep":[]},
    "Array_Voltage":        {"scale":1/(2**15), "bias":0, "round":2, "limit":[], "scale_dep":[[1,"Array_Voltage"],[1,"V_PU"]], "bias_dep":[]},
    "Battery_Current":      {"scale":1/(2**15), "bias":0, "round":2, "limit":[], "scale_dep":[[1,"Battery_Current"],[1,"I_PU"]], "bias_dep":[]},
    "Array_Current":        {"scale":1/(2**15), "bias":0, "round":2, "limit":[], "scale_dep":[[1,"Array_Current"],[1,"I_PU"]], "bias_dep":[]},
    "Output_Power":         {"scale":1/(2**17), "bias":0, "round":2, "limit":[], "scale_dep":[[1,"Output_Power"],[1,"V_PU"],[1,"I_PU"]], "bias_dep":[]},
    "Input_Power":          {"scale":1/(2**17), "bias":0, "round":2, "limit":[], "scale_dep":[[1,"Input_Power"],[1,"V_PU"],[1,"I_PU"]], "bias_dep":[]}
    }
```

Figure 18 Example of `***self._memory_dict*** ` and `***self._extra_calc*** `

2.3.1. "Read" Command

The flow of the program during "read" command can be seen in **Figure 19**. The ***send_command*** function enclose the whole procedure from start to end. At first, the read addresses from the main program (`***main_modbus.py***`) are shifted accordingly if necessary (see **Table 3**). Next, the ***handle_dependency*** function checks whether any dependent variables (from `***self._extra_calc***`) are in that list, then replace them with their dependencies (from `***self._memory_dict***`). After that, the ***reading_sequence*** function is in charge to call other functions to complete the Modbus communication, from sending request message to the device/slave, up to calculating and saving the the reasponse into the object's attribute.

Next, the ***save_read*** function will scale and calculate the response data and save them into the object's attribute based on the corresponding name (*string*) (see **Table 4**). Once the dependencies are complete, the ***handle_extra_calculation*** function will calculate all other dependent variables. Some additional functions such as get_compile_dimension, create_copy_of_compile, and

copy_value_to_compile are necessary for the ***compile*** type (see **Table 5**). Once it is done, the main program can access the object's attribute directly, or through ***map_read_attr*** method (see **Table 2**). An example of the whole procedure can be seen in **Table 6**.

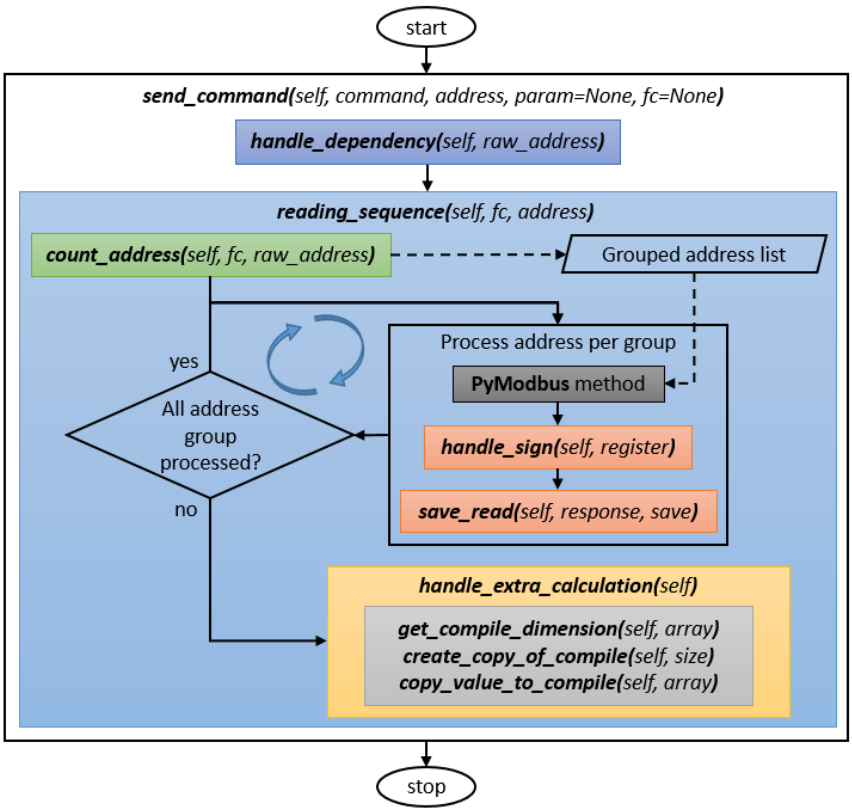

Figure 19 Modbus Device-Specific Library's flowchart during "read" command

2.3.2. "Write" Command

The flow of the program during "write" command can be seen in **Figure 20**. The ***send_command*** function enclose the whole procedure from start to end. At first, the write addresses from the main program (`***main_modbus.py***`) are shifted accordingly if necessary (see **Table 3**). If the read address is is a *string*, then it will obtain the memory address from `***self._memory_dict***`. If the function code to be used and the parameter to be written are not defined yet, it will also obtain the default values from `***self._memory_dict***`. After that, the ***writting_sequence*** function is in charge to call other functions to complete the Modbus communication.

Inside the ***writting_sequence*** function, the ***handle_multiple_writting*** function will parse the parameters to be written into hexadecimal format based on the address increment. The parameters to be written mus be transformed into $4 \times n$ digit in hexadecimal with n being the address increment. This value is then be parsed into 4 digits hexadecimal which equals 2 bytes serial data because Modbus protocol only send and read this data 1 byte at a time. An example of this process is shown in **Table 7**. Next, depending on what the function code is, a ***write_register*** function (*fc* = 0x06) or

write_registers function (*fc* = 0x010) from the PyModbus library will be used to write parameter on the corresponding Modbus address. Remember that function code 0x06 only accepts single value to be written, while function code 0x10 allows multiple writtings (in *array* format) on the adjacent addresses after the first writting address.

Table 6 Example of Modbus "read" Command Procedure

FUNCTION	INPUT*	OUTPUT*	EXAMPLE
send_command	fc $address_{main}$	fc $address_{shifted}$	None ["aC_poWer", 14, 0x000E]
handle_dependency	$address_{shifted}$	$address_{mdict}$	["Volt", "Amp", 14, 0x000E]
reading_sequence	fc $address_{mdict}$	-	-
count_address	fc $address_{mdict}$	fc $address_{order}$ $save_{order}$	0x04 [[0x0000, 0x0002], [0x000E]] [["Volt", "Amp"], ["PF"]]
PyModbus** (read_holding_registers, read_input_registers)	fc $address_{order}$	$response_{order\_raw}$	[[101, 65533], [90]]
handle_sign	$response_{order\_raw}$	$response_{order}$	[[101, -17], [90]]
save_read	$response_{order}$ $save_{order}$	$self.attribute_{mdict}$	self.Volt = 101 self.Amp = -1.7 self.PF = 0.91
handle_extra_calculation	$self.attribute_{mdict}$	$self.attribute_{xcalc\_dep}$	self.AC_Power = -156.25
get_compile_dimension ***create_copy_of_compile*** ***copy_value_to_compile***	$self.attribute_{mdict}$	$self.attribute_{xcalc\_comp}$	Self.Compile_Example = [101, -1.7, 0.9, -156.25]

* The variable name maybe different with what was written in the library
** The function code (*fc* | 0x03 or 0x04) determines which PyModbus method is used

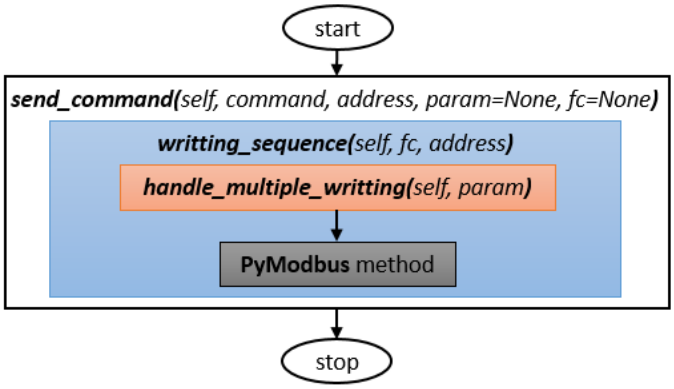

Figure 20 Modbus Device-Specific Library's flowchart during "write" command

Table 7 Example of Parameter Transformation in ***handle_multiple_writing*** Function

VARIABLE*	EXAMPLE
$param_{dec}$ $param_{hex}$	[-205, 132546, 11] [-0xCD, 0x205C2, 0xB]
hex_param**	[ffffff33, 000205C2, 0000000B]
$params_{hex}$ $params_{dec}$	[0xffff, 0xff33,0x0002, 0x05C2,0x0000, 0x000B] [65535, 65331, 2, 1474, 0, 11]
Modbus protocol*	[0xff|0xff, 0xff|0x33, 0x00|0x02, 0x05|0xC2, 0x00|0x00, 0x00|0x0B]

* The variable name maybe different with what was written in the library
** Example for address increment of 2, meaning a 4x2=8 digits of hexadecimal
*** Automatically handled by PyModbus method (***write_register*** and ***write_registers***)

Chapter 3. CANbus System

3.1. Script Installation

After the RS485/CAN Hat module (hardware) is installed, power up the RaspberryPi and copy the folder `***canbus_code***` to Home Folder directory in `***/home/<user>/NIW***` or `***~/NIW***` as shown in **Figure 21**. Next, open terminal and run `***init_canbus.bash***` by executing the command in **Figure 22**. Continue by following the instructions on the terminal.

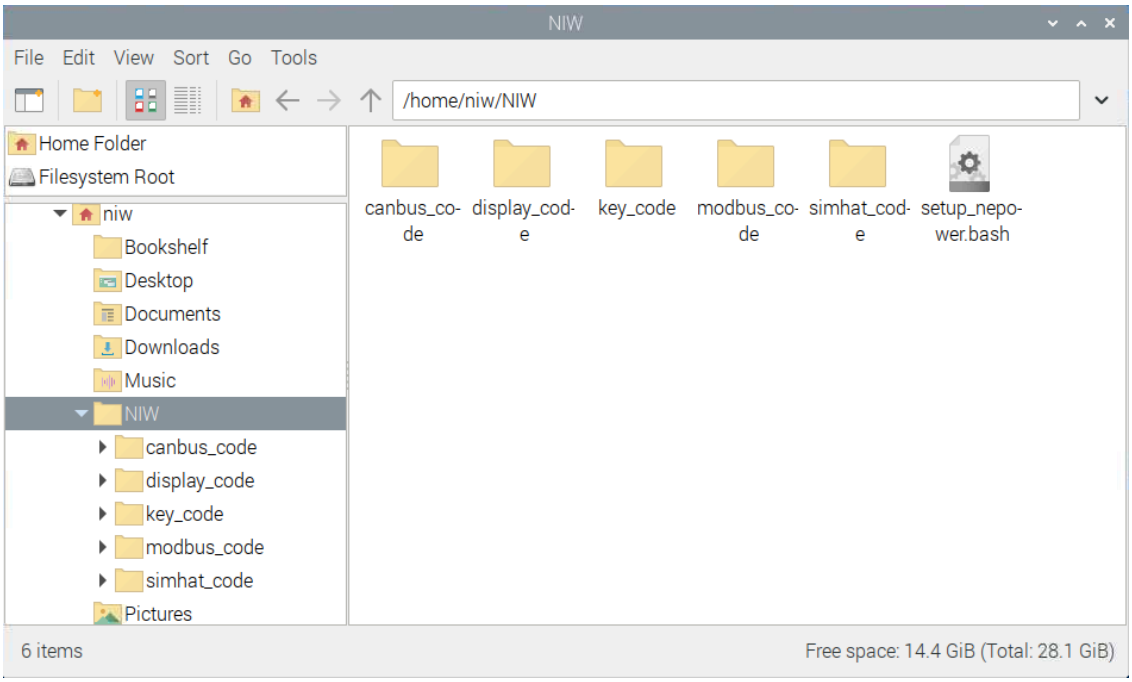

Figure 21 CANbus system installation script directory

```
sudo bash canbus_code/init_canbus.bash
```

Figure 22 Bash command to install the CANbus system

The script will automatically install the necessary package to enable the communication through RS485/CAN Hat module. The installation is complete when you get the message on your terminal as shown in **Figure 23**.

```
============================================================

Installation of CANbus system is finished

Please reboot the RaspberryPi, just in case :)

============================================================
```

Figure 23 Message indicating CANbus system is installed

3.2. Program Architecture

The code is written in Object Oriented Programming (OOP) on Python environment with python-can library as its foundation. The main program is a script called `***main_canbus.py***` inside the `***canbus_code***` folder, while the device-dependent scripts are located in `***canbus_code/lib***` folder. There is also a script called `***query.py***` that is meant for writting/defining specific functions that will be used in the most main programs, not limited to CANbus system only. More about `***query.py***` will be explained in **Chapter 4**. **Figure 24** shows the example of CANbus system's directory.

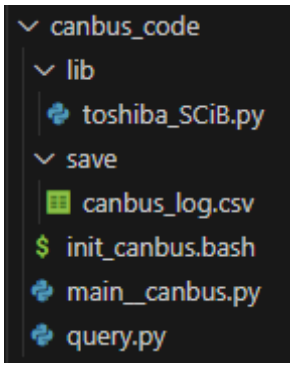

Figure 24 Example of CANbus system's directory

Overall, the program architecture is similar with Modbus Sytem explained on previous chapter. For any projects using the CANbus system, simply create a copy of the `***canbus_code***` folder, replace the scripts inside `***canbus_code/lib***` following the Modbus devices used, then modify the `***main_canbus.py***` as necessary. **Figure 25** show the general flowchart of `***main_canbus.py***`.

The `***setup_canbus()***` function is the mandatory function which initialize the CANbus communication and the device-specific libraries. An example of this function, including its input arguments are shown in **Figure 26**. Each device is initialized as an object using appropriate device-specific library and parameters. To ease working with many devices/slaves, all devices are compiled into one array called `***server***`.

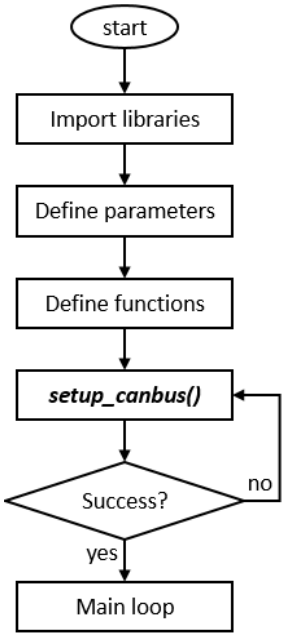

Figure 25 Flowchart of `***main_canbus.py***`

Once initialized, the main program (`***main_canbus.py***`) can receive or dump data (message) from or into the communication bus (CANbus) and it will automatically utilize python-can library to do so. For "receive" command, the device/slave object will calculate and generate a new attribute according to the information that you want. There are three ways to send the "receive" command. First is by passing an array argument filled with list of CAN message's arbitration ID to be read (integer; hexadecimal or decimal). This will configure the program to calculate all data from that message ID. If the ID is not yet available in `***self._can_id***`, then the generated attribute's name will be exactly the ID in hexadecimal (Ex: '0x00A2'). The second method is by passing a more specific array argument filled size-2 array. The index-0 value is the arbitration ID, while the index-1 value is the bit start position. This will configure the program to only calculate that particular data from that message ID. The last method is by passing an array argument filled filled with attribute names to be calculated. The attribute name in this method is case-insensitive. During the time this document is written, the "dump" command has not been integrated into the library yet. The full list of available method can be seen in **Table 8**.

Table 8 Available Methods in the CANbus Device-specific Library

Purpose	Method Syntax
Receive CANbus message ID (*integer*) or calculate custom values (*string*)	obj.send_command(command="receive", address=[*integer,integer,…*]) obj.send_command(command="receive", address=[*string,string,…*]) obj.send_command(command="receive", address=[*[integer,integer],* *[integer,integer],„…*])

Obtain the attribute name ($string_n$) and value ($value_n$) of the object using its CANbus message ID ($integer_n$)	[[$string_1$,$value_1$], [$string_2$,$value_2$],…] = obj.map_rec_attr([$integer_1$, $integer_2$,…])
Reset all attribute values from "receive" command to zero	obj.reset_rec_attr()

* may be omitted if the information is available in `***self._memory_dict***`of the device-specific library

```
# Import library
import can # code packet for CANbus communication
import os
from lib import toshiba_SCiB as battery

# Define CANbus communication parameters
bustype         = 'socketcan' # CANbus interface for Waveshare RS485/CAN Hat module
channel         = 'can0' # location of channel used for CANbus Communication
bitrate         = 250000 # Toshiba SCiB speed of CANbus = 250000
restart         = 100 # the time it takes to restart CANbus communication if it fails (in milisecond)
timeout         = 2 # the maximum time the master/client will wait for response from slave/server (in seconds)
interval       = 1 # the period between each subsequent communication routine/loop (in seconds)

def setup_canbus():
    global bustype, channel, bitrate
    # Configure and bring up the SocketCAN network interface
    os.system('sudo modprobe can && sudo modprobe can_raw') # load SocketCAN related kernel modules
    os.system('sudo ip link set down {}'.format(channel)) # disable can0 before config to implement changes in bitrate settings
    os.system('sudo ip link set {} type can bitrate {} restart-ms {}'.format(channel,bitrate,restart)) # configure can0 & set to 250000 bit/s
    os.system('sudo ip link set up {}'.format(channel)) # enable can0 so configuration take effects
    client = can.interface.Bus(bustype=bustype, channel=channel, bitrate=bitrate)
    bat = battery.node(name='TOSHIBA BATTERY', client=client, timeout=timeout)
    server = bat
    return server
```

Figure 26 Example of `***setup_canbus()***` function

3.3. Device-Specific Library

As shown in **Figure 26**, each CANbus device (slave/server) is an object which is created using a device-specific library which is a Python *class* called `***node***`. At initializations (***__init__***), it created several object attributes (variable/parameter) which are used exclusively in the class's methods (function). These attributes can be categorized generally into 3 categories as explained in **Table 9**. If the information you want is not listed in the `***self._can_id***`, you can edit the library by adding a nested dictionary DataType into the `***self._can_id***`. An example of the dictionary's *key:value* syntax is given in **Figure 27** and is explained in **Table 10**.

Table 9 Private Attributes in `***node***`Class of CANbus Device-Specific Library

Category	Attribute	Purpose
To ease programming	***self._name***	Give individual name to object created using this library
Parameters for python-can communication	***self._client***	Serial port to be used by python-can
	self._timeout	Maximum time to wait for CANbus message (in seconds)
Related to device's Modbus address & data calculation	***self._can_id***	Information related to CANbus device's message ID and data calculation

Table 10 *key:value* Syntax of `***self._can_id*** `

SYNTAX	“attr_name”:{“id”:id,“start”:start,“end”:end,“scale”:scale,“bias”:bias,“round”:round}	
Key	**Value (DataType)**	**Notes**
attr_name	*dictionary*	The calculated data will be saved into the attribute named ***self.attr_name***
id	*integer*	Message arbitration ID of CANbus device
start	*integer*	The start byte of particular information in the CANbus messege (ID)
end	*integer*	The stop byte of particular information in the CANbus messege (ID)
scale	*float/integer*	Used to calculate the data from raw data: $data_{\ calculated} = (data_{\ raw} - bias) \times scale$
bias	*float/integer*	
round	*integer*	The number of digits behind decimals of calculated data (rounded)

```
# Library of CANbus Arbitration ID
self._can_id = {
    "Power_On_Time_1"   :{"id":0x050, "start":1, "end":4, "scale":0.1, "bias":0, "round":1}, # in seconds
    "Power_On_Time_2"   :{"id":0x070, "start":1, "end":4, "scale":0.1, "bias":0, "round":1}, # in seconds
    "IO_Signal_Status_1":{"id":0x050, "start":5, "end":5, "scale":0.1, "bias":0, "round":1}, # per bit value
    "IO_Signal_Status_2":{"id":0x070, "start":5, "end":5, "scale":0.1, "bias":0, "round":1}, # per bit value
    "Module_Address_1"  :{"id":0x050, "start":5, "end":5, "scale":0.1, "bias":0, "round":1},
    "Module_Address_2"  :{"id":0x070, "start":5, "end":5, "scale":0.1, "bias":0, "round":1},

    "Capacity_mAh"      :{"id":0x053, "start":1, "end":2, "scale":1, "bias":0, "round":2}, # in mAh
    "SOC"               :{"id":0x053, "start":3, "end":3, "scale":1, "bias":0, "round":0}, # in %

    "Temperature_1"     :{"id":0x055, "start":3, "end":4, "scale":0.1, "bias":0x8000, "round":1}, # in Celcius
    "Temperature_2"     :{"id":0x075, "start":3, "end":4, "scale":0.1, "bias":0x8000, "round":1}, # in Celcius

    "Module_Current_1"  :{"id":0x056, "start":1, "end":2, "scale":0.01119, "bias":0x8000, "round":2}, # in Ampere
    "Module_Current_2"  :{"id":0x076, "start":1, "end":2, "scale":0.01119, "bias":0x8000, "round":2}, # in Ampere
    "Module_Voltage_1"  :{"id":0x056, "start":3, "end":4, "scale":4.8832/1000, "bias":0, "round":2}, # in Volts
    "Module_Voltage_2"  :{"id":0x076, "start":3, "end":4, "scale":4.8832/1000, "bias":0, "round":2} # in Volts
    }
```

Figure 27 Example of `***self._can_id*** `

3.3.1. “Receive” Command

The flow of the program during “read” command can be seen in **Figure 28**. The ***send_command*** function enclose the whole procedure from start to end. At first, the read addresses from the main program (`***main_canbus.py***`) are…. After that, the ***reading_sequence*** function is in charge to call other functions to complete the Canbus communication, from sending request message to the device/slave, up to calculating and saving the the reasponse into the object’s attribute.

Next, the ***save_read*** function will scale and calculate the response data and save them into the object’s attribute based on the corresponding name (*string*) (see **Table 10**). Once it is done, the main program can access the object’s attribute directly, or through ***map_read_attr*** method (see **Table 8**). An example of the whole procedure can be seen in **Table 11**.

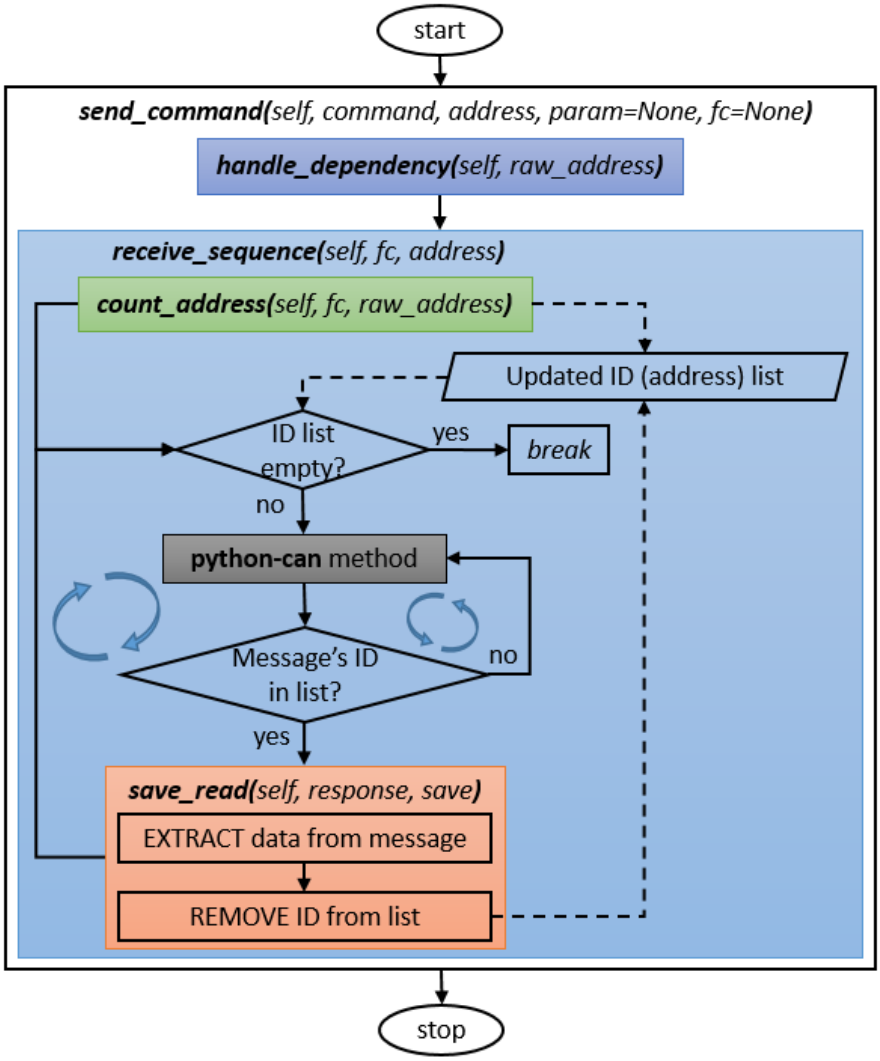

Figure 28 CANbus Device-Specific Library's flowchart during "receive" command

Table 11 Example of CANbus "receive" Command Procedure

FUNCTION	INPUT*	OUTPUT*	EXAMPLE
send_command	fc $address_{main}$	fc $address_{shifted}$	None ["aC_poWer", 14, 0x000E]
receive_sequence	fc $address_{mdict}$	-	-
count_address	fc $address_{mdict}$	fc $address_{order}$ $save_{order}$	0x04 [[0x0000, 0x0002], [0x000E]] [["Volt", "Amp"], ["PF"]]
python-can**	fc $address_{order}$	$response_{order\_raw}$	[[101, 65533], [90]]
dump_sequence	–	–	-
save_read	$response_{order}$ $save_{order}$	$self.attribute_{mdict}$	self.Volt = 101 self.Amp = -1.7 self.PF = 0.91

* The variable name maybe different with what was written in the library

** The function code (*fc*) determines which python-can method is used, but now it is only able to read

Chapter 4. Additional Information

4.1. Query.py

The `***query.py***` script holds some specific functions that can be used in most main programs to ease code maintain. It also can be used by any Python script as a library and user can add more functions if needed. The functions in `***query.py***` is separated from the main program to improve code readability and maintainability. The full list of this functions are listed in **Table 12**. Another main purpose of the `***query.py***` script is in deploying a robust interaction with MySQL database, which includes backing up the data if it fials connecting to MySQL server (ex: when there is no internet connection). This process is illustrated in **Figure 29.** An example of how to utilize the `***query.py***` functions to interact with MySQL database through the main program is shown in **Figure 30.**

Table 12 General Functions in `***query.py***` Script

FUNCTION	PURPOSE
debugging	Print the backend process of Python library, used expecially for debugging the PyModbus method
handle_timeout	Raise "TimeoutError" exception, used especially for handling MySQL query execution
get_cpu_temperature	Obtain the temperature of RaspberryPi's CPU, used for device monitoring
print_response	Print attributes of an Object DataType (except private attributes), used especially for device-specific library
strval	Convert an array into string separated by space character, used in saving array values in MySQL or CSV, utilized by ***log_in_csv*** and ***connect_mysql*** function. It doeas not work with nested array
log_in_csv	Save a data in CSV file which may be used later on, either simply for data logging or for backup when there is no internet connection
connect_mysql	Execute MySQl query using PyMySQL library, may be used either with INSERT or SELECT command.
retry_mysql	Logic function for updating the MySQL database using all the data in CSV and remove the corresponding rows in the CSV file. It stops only if the CSV file is empty or if it fails connecting to MySQL database. It inherently utilize the ***connect_mysql*** function.
limit_db_rows	MySQl query to limit the database size by deleting the rows which are over the limit. It inherently utilize the ***connect_mysql*** function.
get_updown_time	MySQl query to calculate culmulative uptime and downtime. Modify this function if necessary (different table's column name). It inherently utilize the ***connect_mysql*** function.

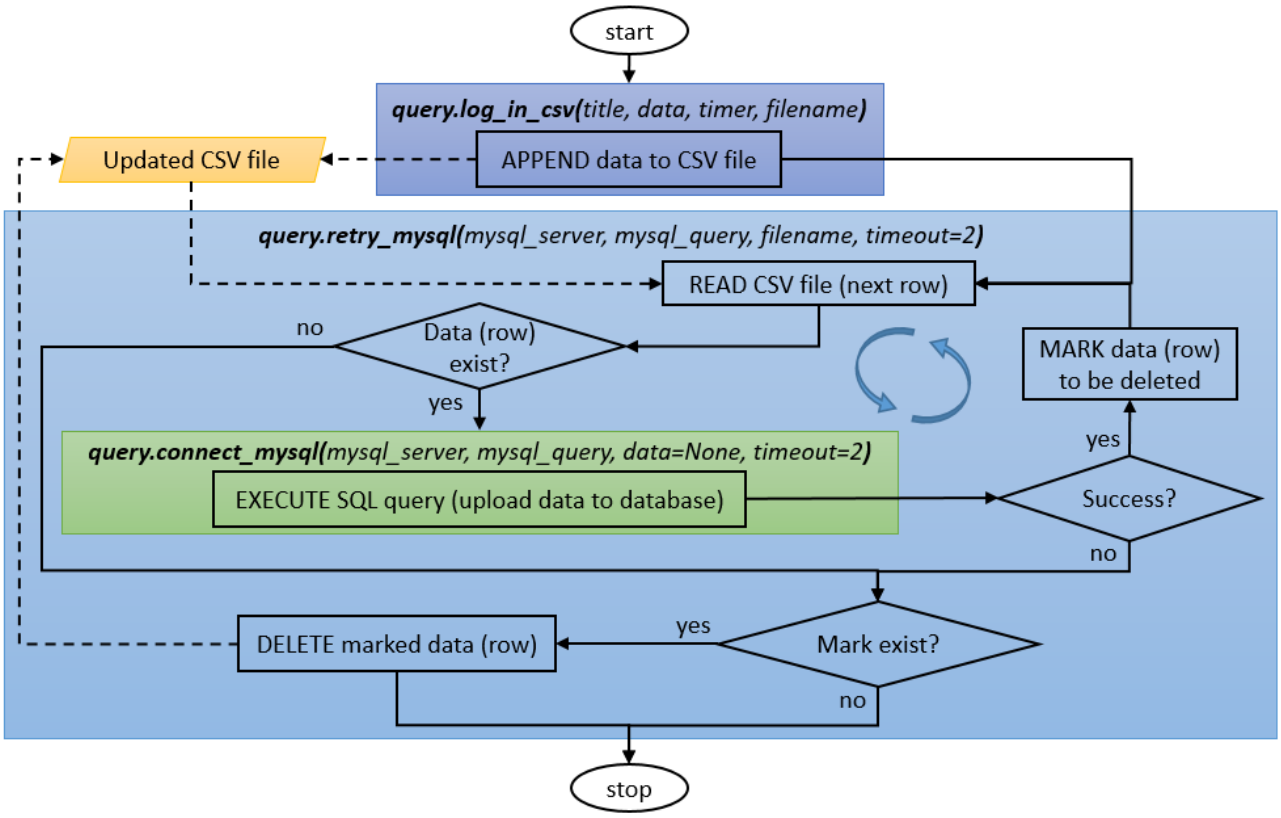

Figure 29 Flowchart of procedure in backup and update to database

```
import query
import datetime # RTC Real Time Clock
from lib import SIM7600_GNSS as gnss

# Define MySQL Database parameters
mysql_server    = {"host":"10.4.171.204",
                   "user":"pi",
                   "password":"raspberrypi",
                   "db":"test",
                   "table":"diff_gps",
                   "port":3306}
mysql_timeout   = 3 # the maximum time this device will wait for completing MySQl query (in seconds)
gps = gnss.node('/dev/ttyAMA0', 'Forklift A')
timer = datetime.datetime.now()

## Get RaspberryPi's temeperature
cpu_temp = query.get_cpu_temperature()

## Calculate RaspberryPi's culmulative uptime and downtime from MySQL database
[bootup_time, uptime, total_uptime, downtime, total_downtime] = query.get_updown_time(mysql_server, timer, mysql_timeout)

## Get data from MySQL database using SELECT command
select_query = ("SELECT latitude, longitude, hdop, satelites FROM {} ORDER BY id DESC LIMIT 1".format(mysql_server["table"]))
base_data = query.connect_mysql(mysql_server,select_query,timeout=mysql_timeout)

## Backup the data then update MySQL database using INSERT command, the array value is automatically converted into string
title = ["RPi_Temp","Lat","Lon","base_data","datetime_now","total_uptime"]
data = [cpu_temp, gps.Latitude, gps.Longitude, base_data, timer.strftime("%Y-%m-%d %H:%M:%S"), total_uptime]

insert_query = ("INSERT INTO `{}` ({}) VALUES ({})".format(mysql_server["table"],
                                                           ",".join(title),
                                                           ",".join(['%s' for _ in range(len(title))])))
filename = 'modbus_log.csv'
query.log_in_csv(title ,data, timer, filename)
query.retry_mysql(mysql_server, insert_query, filename, mysql_timeout)
```

Figure 30 Example use of `***query.py***`script in the main program

4.2. Get_usb.bash

The '**get_usb.bash**' scripts contains specific commands to find USB path for connection between Raspberry Pi and the removable devices. It is applied to connect to Modbus and Canbus Communication, find the Simhat/modem device, and also device key for authentication purpose.

There are 2 algorithm here, 'Path finder' or 'Device Checker'. 'Path finder' is used to find a devices path with certain name and return the value of the path, you can see the example scripts in **Figure 31**. 'Device Checker' is used to check if device that contains specific files is inserted or not, you can see the example scripts in **Figure 32**.

```
# Find the line with specific USB name
id=$1
line=$(ls -l /dev/serial/by-id | grep "$id")

# Extract the 'ttyUSB*' part from the line
path=$(echo "$line" | awk '{print $NF}')

# Check if the device name was found
if [ -n "$path" ]; then
filename=$(basename "$path")
device_name="${filename%.*}"
sudo chmod a+rw /dev/$device_name
echo "/dev/$device_name"
else
sudo chmod a+rw $id
echo $id
fi
```

Figure 31 Path Finder get_usb.bash scripts

```
directory="/media/$(logname)"
result=0

# Check if the directory is not empty
if [ "$(ls -A $directory)" ]; then
# Iterate over subdirectories
for subdirectory in "$directory"/*; do
# Check if iot_key.txt file exists in the subdirectory
if [ -f "$subdirectory/iot_key.txt" ]; then
result=1
fi
done
if [ result == 0 ]; then
echo "$directory/devicenotfound"
else
echo "$subdirectory"
fi
fi
exit
```

Figure 32 Device Checker get_usb.bash scripts

4.3. GPS System

The

4.4. USB IoT Key

The

4.5. HMI Program

The