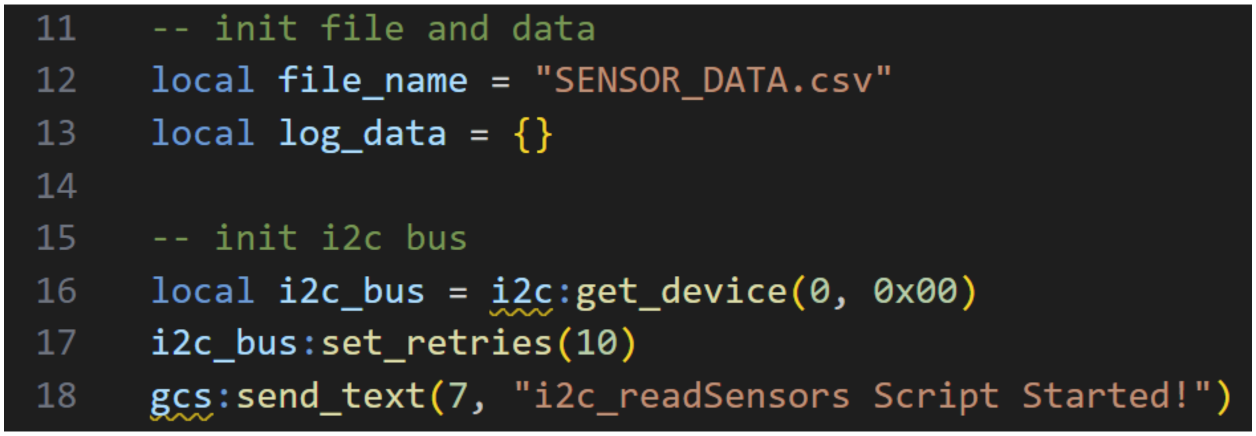
Ryan Prince
UAVLab i2c_readSensors Documentation

This document intends to explain the code written to read from the given sensors that are uniquely addressable. Part number of the sensors used is ELVH-L02D HAAH-C N5A4.

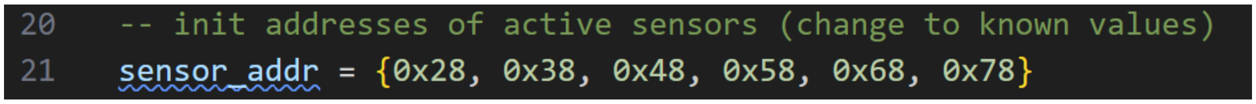
Starting from the top down:



```lua
-- init file and data
local file_name = "SENSOR_DATA.csv"
local log_data = {}

-- init i2c bus
local i2c_bus = i2c:get_device(0, 0x00)
i2c_bus:set_retries(10)
gcs:send_text(7, "i2c_readSensors Script Started!")
```

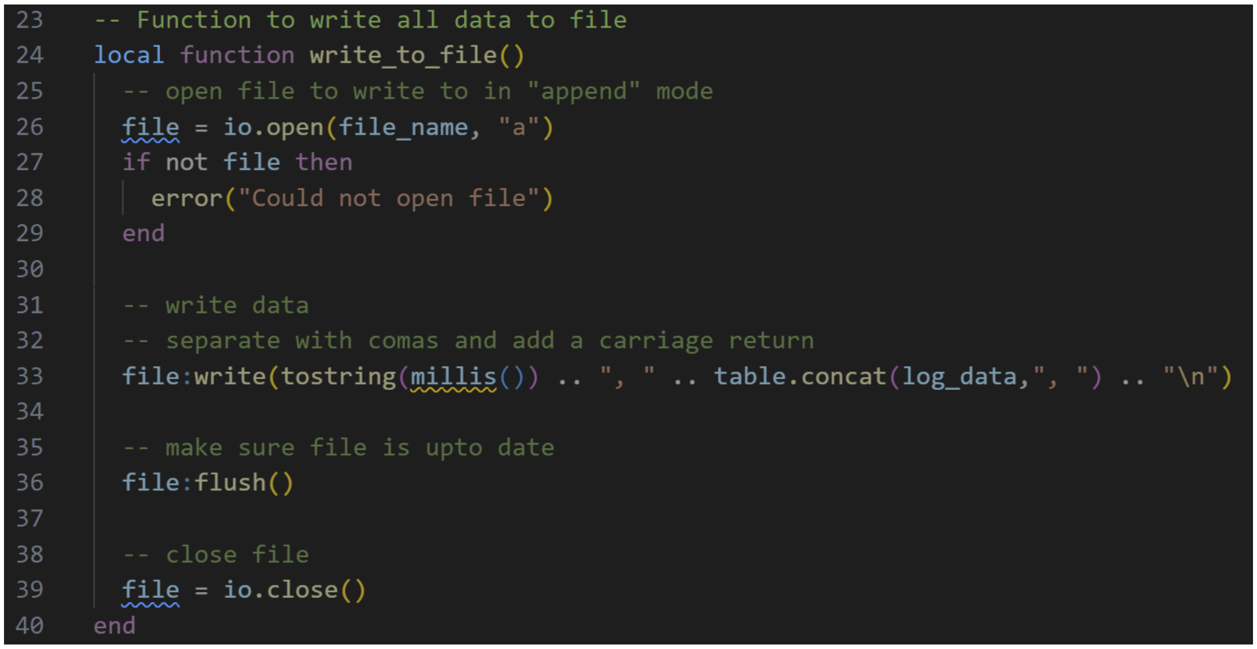
These lines set up variables that will be used throughout the code. “file_name” is the name of the file that gets written to. “log_data” is an array that will store the data values read from the sensors.

“i2c_bus” is the data stream that we will be communicating on to talk to the sensors. Setting retries is just in-case the bus isn’t immediately detected. And then a debug message is sent to the console to let us know the script is running.



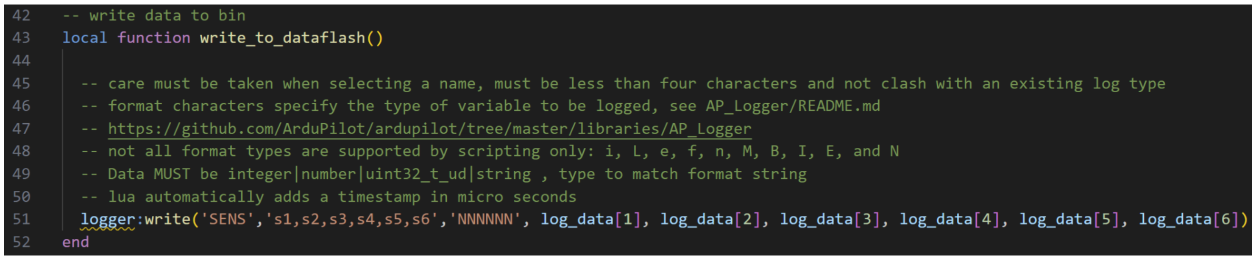
```lua
-- init addresses of active sensors (change to known values)
sensor_addr = {0x28, 0x38, 0x48, 0x58, 0x68, 0x78}
```

This array will store the addresses of the sensors that need to be read from. This will vary depending on the specific sensor used. Note: these must be unique in this program. The address is printed on the side of the sensors (ELVH).



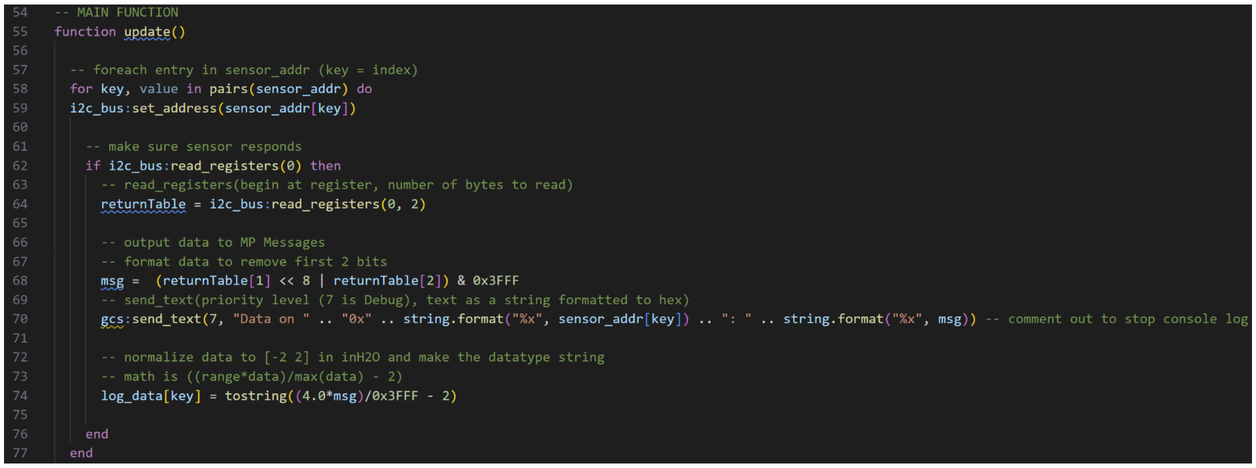
```lua
-- Function to write all data to file
local function write_to_file()
  -- open file to write to in "append" mode
  file = io.open(file_name, "a")
  if not file then
    error("Could not open file")
  end

  -- write data
  -- separate with comas and add a carriage return
  file:write(tostring(millis()) .. ", " .. table.concat(log_data,", ") .. "\n")

  -- make sure file is upto date
  file:flush()

  -- close file
  file = io.close()
end
```

This function writes to the previously defined "file_name." This file will quickly grow in size depending on the sample time used. In the code it is grayed out because I stopped using it once I finished the .bin file writing. This should only be used for debugging purposes.



```lua
-- write data to bin
local function write_to_dataflash()

  -- care must be taken when selecting a name, must be less than four characters and not clash with an existing log type
  -- format characters specify the type of variable to be logged, see AP_Logger/README.md
  -- https://github.com/ArduPilot/ardupilot/tree/master/libraries/AP_Logger
  -- not all format types are supported by scripting only: i, L, e, f, n, M, B, I, E, and N
  -- Data MUST be integer|number|uint32_t_ud|string , type to match format string
  -- lua automatically adds a timestamp in micro seconds
  logger:write('SENS','s1,s2,s3,s4,s5,s6','NNNNNN', log_data[1], log_data[2], log_data[3], log_data[4], log_data[5], log_data[6])
end
```

This function writes to the .bin file using the logger. The data type defined in the 3rd parameter must match the actual data type of the data. In this case, the "log_data" is a string and the parameter "N" is telling the logger the data is a 16 character char value. More about data types can be found here:
https://github.com/ArduPilot/ardupilot/tree/master/libraries/AP_Logger

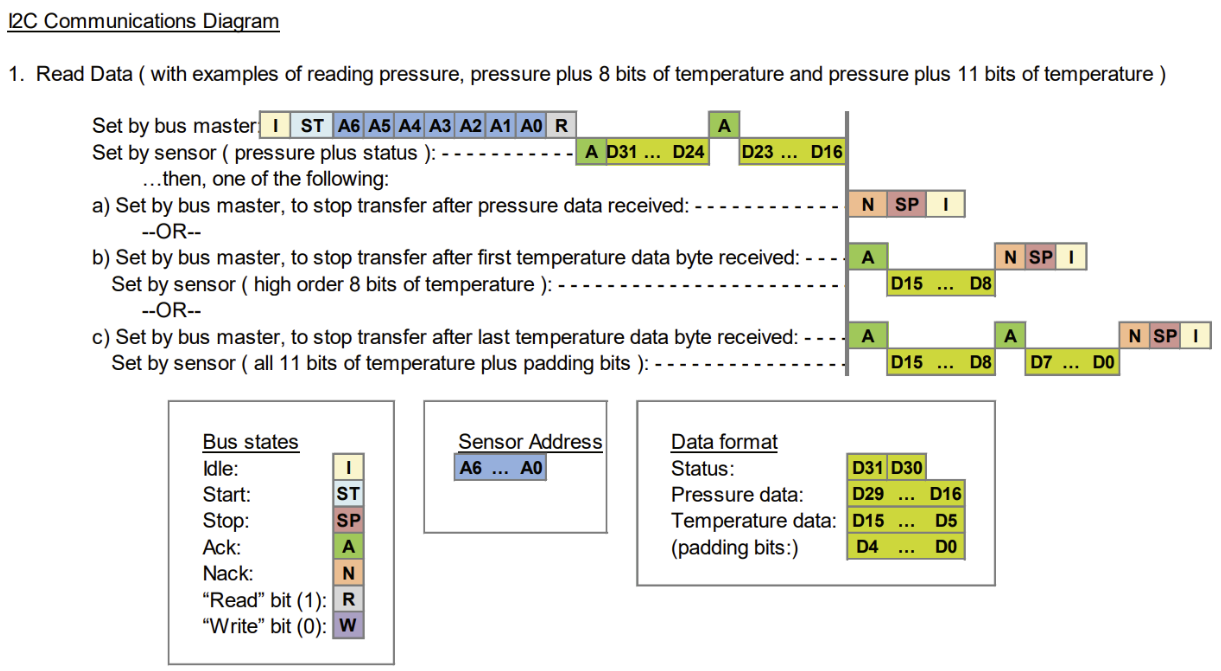


```lua
-- MAIN FUNCTION
function update()

  -- foreach entry in sensor_addr (key = index)
  for key, value in pairs(sensor_addr) do
  i2c_bus:set_address(sensor_addr[key])

    -- make sure sensor responds
    if i2c_bus:read_registers(0) then
      -- read_registers(begin at register, number of bytes to read)
      returnTable = i2c_bus:read_registers(0, 2)

      -- output data to MP Messages
      -- format data to remove first 2 bits
      msg =  (returnTable[1] << 8 | returnTable[2]) & 0x3FFF
      -- send_text(priority level (7 is Debug), text as a string formatted to hex)
      gcs:send_text(7, "Data on " .. "0x" .. string.format("%x", sensor_addr[key]) .. ": " .. string.format("%x", msg)) -- comment out to stop console log

      -- normalize data to [-2 2] in inH2O and make the datatype string
      -- math is ((range*data)/max(data) - 2)
      log_data[key] = tostring((4.0*msg)/0x3FFF - 2)

    end
  end
```

Now we are on to the main function which implements the previous functions as well as communicating with the i2c bus.

We first want to loop through all of the addresses of the sensors that were defined in "sensor_addr." We will then tell the bus we want to talk to each address.

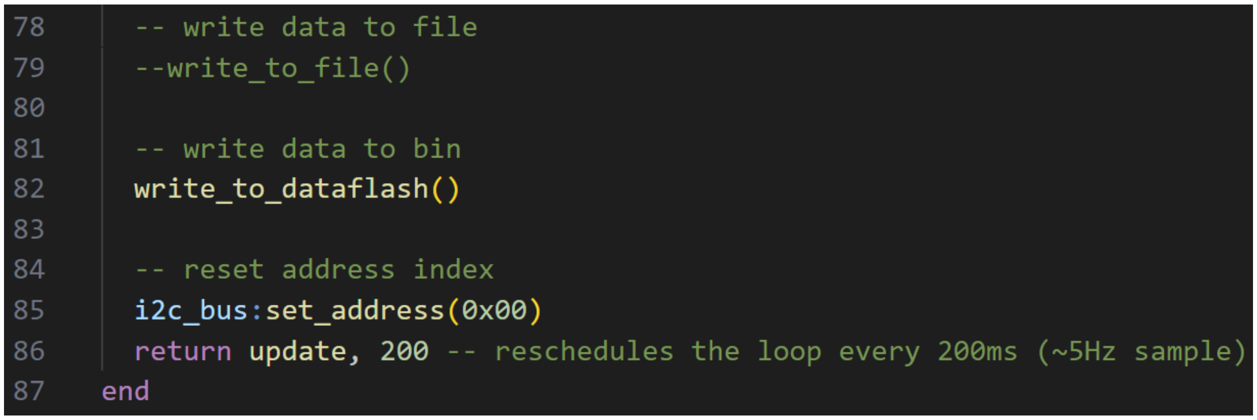
Then we want to make sure the sensor responds before continuing. If it does, we want to get the first 2 bytes, which contains all of the temperature data as well as 2 flags. To do this we set up a variable that is a table of the data divided into bytes. We need to use the first two bytes, starting with no offset, hence the call read_registers(0, 2). The two flags will be ignored/cleared next. How the data is formatted can be seen below:


I2C Communications Diagram
1. Read Data ( with examples of reading pressure, pressure plus 8 bits of temperature and pressure plus 11 bits of temperature )
Set by bus master: I ST A6 A5 A4 A3 A2 A1 A0 R A
Set by sensor ( pressure plus status ): A D31 … D24 D23 … D16
…then, one of the following:
a) Set by bus master, to stop transfer after pressure data received: N SP I
--OR--
b) Set by bus master, to stop transfer after first temperature data byte received: A N SP I
Set by sensor ( high order 8 bits of temperature ): D15 … D8
--OR--
c) Set by bus master, to stop transfer after last temperature data byte received: A A N SP I
Set by sensor ( all 11 bits of temperature plus padding bits ): D15 … D8 D7 … D0
Bus states
Idle: I
Start: ST
Stop: SP
Ack: A
Nack: N
"Read" bit (1): R
"Write" bit (0): W
Sensor Address
A6 … A0
Data format
Status: D31 D30
Pressure data: D29 … D16
Temperature data: D15 … D5
(padding bits:) D4 … D0

So in our case we want D29-D16 which is 14 bits so 2 bytes are needed. D31-D30 we don't care about so we will AND the data read with 0x3FFF or (0011 1111 1111 1111) which retains all data except the first 2 bits.

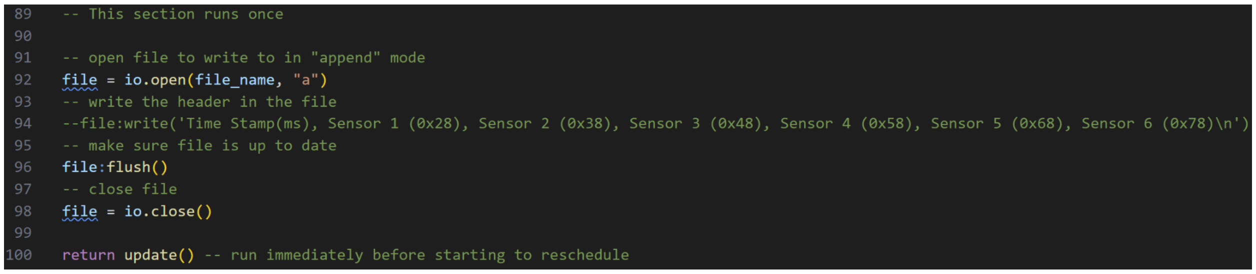
Next we will output the data we just got, in HEX format to the console log. This is used to make sure the sensors are reading and we are interpreting the data correctly.

Lastly, we will normalize the data to the dynamic range we wanted. Based on the datasheet, the range of the sensors used is [-2 2] inH2O. This data will then be casted to a string so it can be written to the .bin file.



```lua
    -- write data to file
    --write_to_file()

    -- write data to bin
    write_to_dataflash()

    -- reset address index
    i2c_bus:set_address(0x00)
    return update, 200 -- reschedules the loop every 200ms (~5Hz sample)
end
```

Then we of course need to write to the .bin file, calling the function made before and inputting the string data we just made. Here, the option to output to a file is given but is commented due to not being necessary. Then we will reset the current address we are connected to and run the loop again.



```lua
-- This section runs once

-- open file to write to in "append" mode
file = io.open(file_name, "a")
-- write the header in the file
--file:write('Time Stamp(ms), Sensor 1 (0x28), Sensor 2 (0x38), Sensor 3 (0x48), Sensor 4 (0x58), Sensor 5 (0x68), Sensor 6 (0x78)\n')
-- make sure file is up to date
file:flush()
-- close file
file = io.close()

return update() -- run immediately before starting to reschedule
```

Again, this is related to writing to a file, which is not needed but to explain it, we open the file, write a header to it so when data is written it has a label to it. Then we will flush it, which makes sure we have the most recent file open. Lastly, we close the file as we no longer need access to it. Closing the file is just good practice but not essential to making the program work.

References

ELVH Sensor datasheet: https://www.allsensors.com/datasheets/DS-0376_Rev_A.pdf
Ardupilot logging info: https://github.com/ArduPilot/ardupilot/tree/master/libraries/AP_Logger
Ardupilot lua function definition:
https://github.com/ArduPilot/ardupilot/blob/master/libraries/AP_Scripting/docs/docs.lua
Cube orange documentation:
https://docs.cubepilot.org/user-guides/autopilot/the-cube-module-overview