Table of Contents

- Code Overview
  - Global Variables
  - Serial Initialization
  - Update Function
  - Verifying Checksum
  - Parsing Data
  - Logging
  - Important Notes
- Important Links

Code Overview

Over view of the `serial_pth.lua` script for decoding and logging data from the PTH serial sensor

> **NOTE**: This document is in the order that the code is ran, not in the order that it is organized in the file.

Global Variables

```lua
-- variable to count iterations without getting message
local loops_since_data_received = 0

-- variables to count number of bytes available on serial line
local previous_n_bytes = 0
local n_bytes = 0


-- error type table
local error_list = {
  "No data received",      -- 1
  "Checksum fail",         -- 2
  "Data parsing fail",     -- 3
}
```

The `loops_since_data_received` variable stores how many loops since a new byte of data has been received on the serial line. If this value goes above a certain count, we can assume that the sensor is most likely disconnected, and can report and log an error

The `n_bytes` and `previous_n_bytes` variables store the amount of bytes available to read from the serial line, and the bytes available to read on the last iteration of the script respectively. We can use this data to check and see if the sensor has stopped sending data in the middle of a transmission. Which cannot be picked up if we only check if `n_bytes` is zero.

The `error_list` table holds a list of possible errors we can receive. We use this list to pass through the kind of error we are experiencing to the `log_error()` function to be logged in the BIN file for later reference.

Serial Initialization

```
1  -- initialize serial connection
2  local BAUD_RATE = 9600
3
4  -- find the serial first (0) scripting serial port instance
5  -- SERIALx_PROTOCOL 28
6  local PORT = assert(serial:find_serial(0),"Could not find Scripting
       Serial Port")
7
8  -- begin the serial port
9  PORT:begin(BAUD_RATE)
10 PORT:set_flow_control(0)
```

This sets up the serial connection between the PTH and the flight controller. A baudrate of 9600 is used since this is the speed the sensor operates at.

After we set the baudrate, we then find the serial port that is currently open and using protocol 28, which is the ArduPilot protocol for scripting.

> **NOTE**: make sure to set the correct baudrate of 9600 and protocol 28 on the serial line being used to ensure the sensor is connected.

If we cannot find a serial port with the scripting protocol, the script throws an error.

The serial initialization is pulled from the ArduPilot Serial Dump example.

Update Function

Once the serial initialization is completed, the script then jumps to the end of the file (since that there is no other code outside of the functions) and runs the following return statement.

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

This statement calls the `update()` function, which is the main function in most ArduPilot lua scripts.

First snippet of the `update()` function.

```
1  function update()
2    previous_n_bytes = n_bytes
3    n_bytes = PORT:available()
4
5    -- If we have received no bytes or have not received any new bytes,
       increment
6    -- the count of loops without data. If it reaches 6 or more
7    -- (250ms * 6 = 1.5sec), then log an error.
8    if (n_bytes <= 0) or (n_bytes == previous_n_bytes) then
9      loops_since_data_received = loops_since_data_received + 1
10     if loops_since_data_received >= 6 then
11       log_error(error_list[1])
12       gcs:send_text(0, "ERROR: PTH has failed to send data")
13       gcs:send_text(0, "Bytes received: " .. tostring(n_bytes))
14     end
15       return update, 250
16   else
17     loops_since_data_received = 0
18   end
19
20   -- If we are in the middle of recieving a message,
21   -- simply wait for the rest of the message to arrive
22   if (n_bytes > 0 and n_bytes < 60)  then
23     return update, 250
24   end
```

We first set the value of `previous_n_bytes` to the current `n_bytes` value, and then read the number of bytes that are available to read on the serial bus.

Once we update `n_bytes`, we check if it is zero. If it is zero, this means that there is no new data on the bus, meaning the sensor is disconnected, or it has not sent its data yet. We also check if the bytes on the previous loop are the same as the current bytes available, which signals to us that the sensor has stopped working in the middle of a data transmission.

If either of these conditions are true, we increment the `loops_since_data_received` counter. We use this variable to keep track of how many times the script has ran without receiving new data. If we reach six or more iterations without new data, we log an error with the `log_error()` function (This will be discussed further in the "Logging" section) and send an error out to the Mission Planner output.

```
1  gcs:send_text(0, "ERROR: PTH has failed to send data")
```

The `send_text()` method takes two arguments. The first being the priority/type of message we are sending. In this case, we use priority `0`, which specifies that this is an error message and needs to be

displayed immediately. The second argument is simply a string that contains the message that will be sent.

As mentioned above, we wait for 6 failed loops before we log an error. We derive this number of failed loops from how often we schedule the `update()` function to run, which is every 250 milliseconds, and how often the PTH sends out its data, which is every second. To ensure that we are not flagging the time in between data transmissions from the sensor as an error, we need to wait more than one second. By waiting for six failed loops, we guarantee that we are waiting at least one and a half seconds before we log an error ($250\text{ms} \cdot 6 = 1500 = 1.5\text{sec}$). This gives the PTH plenty of time to send its data and will prevent us from logging any false errors.

If we get passed the above checks, we then verify if we are in the middle of a data transmission. If we are, we simply return and reschedule the `update()` function since we do not want to process an incomplete data transmission.

Second snippet of the `update()` function.

```
while n_bytes > 0 do
  -- only read a max of 60 bytes in a go
  -- this limits memory consumption
  local buffer = {} -- table to buffer data
  local bytes_target = n_bytes - math.min(n_bytes, 60)
  while n_bytes > bytes_target do
    table.insert(buffer,PORT:read())
    n_bytes = n_bytes - 1
  end
```

If we pass the above checks, we know that we have received a full message from the Samamet and we can now begin to process the data.

We first take the amount of bytes available on the line and limit the amount we will read to 60 bytes. We do this to ensure we only process one message at a time if we were to have multiple messages on the serial bus (which very unlikely given the previous checks).

We then loop, reading the bytes from the serial line, appending them to our table called `buffer`.

> The primary loop that reads the data from the serial line is also pulled from the ArduPilot Serial Dump example.

Third snippet of the `update()` function.

```
    local data = string.char(table.unpack(buffer))
    -- check if checksum is valid
    if (verify_checksum(data)) then
      -- make sure that data is logged correctly
      if not (parse_data(data)) then
        log_error(error_list[3])
```

```
7           gcs:send_text(0, "ERROR: PTH data was not successfully parsed
               or not written to BIN file correctly!")
8           gcs:send_text(0, "Incoming string: " .. data .. string.format("
                size: %d", #data))
9         end
10      else
11        log_error(error_list[2])
12        gcs:send_text(0, "ERROR: PTH Data failed checksum, check sensor!"
             )
13        gcs:send_text(0, "Incoming string: " .. data .. string.format("
             size: %d", #data))
14      end
15    end
16
17    return update, 250 -- reschedules the loop every 250ms
18  end
```

Once we have placed the message from the serial line into our `buffer` table, we can concatenate it into a string in the variable `data`.

First we verify that the checksum provided with the message, is correct. This occurs in the `verify_checksum()` function, the specifics of which will be discussed later in the "Verifying Checksum" section.

Below is the message format of the PTH sensor. At the end of the message, after the asterisk, is a two digit hexadecimal number, represented as a string. This is the checksum of the message.

```
1  $UKPTH,000E,098152.5,Pa,23.17,C,22.90,C,42.21,%,22.45,C*4A<CR><LF>
```

To calculate the checksum from the message, we take the "main body" of the message, that is the text inside, but **NOT INCLUDING** the "`$`" and "`*`". The main body of the message can be seen below.

```
1  UKPTH,000E,098152.5,Pa,23.17,C,22.90,C,42.21,%,22.45,C
```

Once we have the "main body" of the message, we can now calculate the checksum. As per the documentation for the sensor:

> The checksum was calculated as the bit-wise exclusive OR of all 8-bit ASCII characters between, but not including, '$' and '*' and displayed as a 2-digit hexadecimal number

An explanation of the exclusive OR operation (XOR) can be found here.

We take each of the 8-bit ASCII characters in the "main body" of the message string, and successively XOR each character with the next one.

In the example above, we start out with the ASCII character U. We then XOR U with the next character in the string, K. After we XOR these two characters together, we take the result of this operation, and

then XOR it with the next character in the messsage, in this case P. We then repeat this process until we reach the end of the string.

Once we calculate the checksum from the message, we can now verify if the message is valid. We do this by comparing the value of the checksum we calculated, to the checksum sent with the message. If the two values do not match, we know that the data contained within the message, or the checksum bytes themselves, are corrupted in some manner.

If we find that the checksum is invalid, we report this error to Mission Planner and do `log_error()` to log an error in the BIN file.

Once the data is verified, we can begin parsing and logging the data. The specifics will be discussed futher in the "Logging" and "Parsing Data" sections.

We start by extracting the "main body" of the message into a string. We then take each section of the message, which is delimited by commas, and place them into a table.

Once we have a table of all of the sections in the message, we then can then extract the data sections and place them into their own table.

We then pass this new table of just data values to the `log_data()` function, which logs the data to the BIN file with names for each piece of data, and their appropriate units.

If the `log_data()` function detects that the input table does not meet the required size of 5 elements, it will return false, and not log the data. the `parse_data()` returns the return value of `log_data()` to `update()`.

When `parse_data()` returns false in the above case, or the other cases the function can detect, `update()` reports to Mission Planner that the data was not successfully and logs and error.

`update()` also will report the string that it read which failed the checks, and report its size. This is primarily for debugging purposes.

```
1  if not (parse_data(data)) then
2    log_error(error_list[3])
3    gcs:send_text(0, "ERROR: PTH data was not successfully parsed or not
        written to BIN file correctly!")
4    gcs:send_text(0, "Incoming string: " .. data .. string.format(" size:
         %d", #data))
5  end
```

If both `verify_checksum()` and `parse_data()` return true, the data that was read from the serial line was successfully logged. We can now continue and reschedule the `update()` function to read the next message.

```
1  return update, 250 -- reschedules the loop every 250ms
```

Here we schedule the update function to return every 250 milliseconds. We schedule the loop to run significantly faster that what the Samamet to prevent cases where we receive more than one message for each run of the `update()` function. Receiving more than one message causes us to log the messages that were recorded around a second apart, but in the log file, show to be around only three milliseconds apart. To prevent this we simply schedule the loop to run faster than the sensor sends out its data, and perform some simple checks (that were discussed earlier) to prevent any false errors from being logged.

Verifying Checksum

Snippet of the `verify_checksum()` function, with comments removed.

```
function verify_checksum(message_string)
  local data_string = message_string:match("%$(.*)%*")

  if data_string == nil then
    return false
  end

  local incoming_checksum = message_string:match("%*([0-9A-F][0-9A-F])"
      )

  if incoming_checksum == nil then
    return false
  end

  incoming_checksum = tonumber(incoming_checksum, 16)

  local checksum = 0x0
  local string_bytes = { data_string:byte(1, #data_string) }
  for i = 1, #string_bytes do
    checksum = (checksum ~ string_bytes[i])
  end

  if checksum ~= incoming_checksum then
    return false
  else
    return true
  end
end
```

We first take the message string and perform a regular expression (regex or regexp) match on the string. Here the Lua regex '`%$(.*)%*`' first looks for a '`$`', once it finds one, it then matches any characters after the '`$`' up until it finds a '`*`'. This extracts the main body of the message, which contains the data that we need to process for logging. An example of a full message can be seen below.

```
$UKPTH,000E,098152.5,Pa,23.17,C,22.90,C,42.21,%,22.45,C*4A<CR><LF>
```

Before we continue, we check if the regex failed, if it has, it will have returned a value of `nil`. We check for this, and if this is true, we return false for the caller to handle.

We then perform another regex on the message string again to extract the checksum. The regex `'%*([0-9A-F][0-9A-F])'` first finds a '*'. After it finds one, it then matches exactly two characters. Since we are matching for a hexadecimal number, the regex will only accept characters in hexadecimal numbers. This includes all digits between zero and nine, and all upper case version of letters between and including A-F.

Once we have extracted the checksum, we verify that the regex was successful by making sure the resulting string is not `nil`. If it is `nil`, we return false for the caller to handle.

If we successfully extraced the checksum value, we then need to convert it to an integer since we cannot compare the string directly with the checksum value we will calculate later. To do this we call the `tonumber()` function. We pass in the string we want convert to a number, and the base of the number we are passing in. In this case with a hexadecimal number, we specify `16`.

Now that we have extracted the main message body and the incoming checksum, we can now calculate the checksum ourselves and verify it is correct.

We start by creating the `checksum` variable to hold our calculated checksum and set it to zero. We do this so we can perform the first XOR with the first character in the string without causing any issues.

We then need to convert the `data_string` variable into an array of bytes. We need to do this for two reasons. One, we need to be able to iterate over the string easily, and two, Lua does not support doing bitwise operations, (such as XOR) on strings or characters directly.

```
local string_bytes = { data_string:byte(1, #data_string) }
```

The above code snippet first takes the first character in the string, and returns its ASCII value. We place this expression inside of a set of curly braces to take all of the ASCII values of the characters in the string and place them in a table.

Once we have done that we can finally calculate the checksum. As mentioned before. We calculate the checksum by simply XORing each character with the result of the previous XOR operation.

Once we have calculated the checksum, we compare it with the incoming checksum. If the two are not the same, we return false, meaning that the data has been corrupted at some point during the transmission. If the two values are the same, we return true, as the data has not been effected and we can continue processing the data.

For information on Lua's regular expressions, you can view these pages:

- Pattern-Matching Functions

- Lua pattern matching

For making patters and regular expressions in Lua, you can use these web tools:

- Lua Patterns Viewer
- Lua Pattern Tester

Parsing Data

Snippet of the `parse_data()` function

```
function parse_data(message_string)
  local data_string = message_string:match("%$(.*)%*")

  if data_string == nil then
    return false
  end

  local data_table = {}

  for str in string.gmatch(data_string, "([^" ..","..  "]+)") do
    table.insert(data_table, str)
  end

  if #data_table ~= 12 then
    return false
  end

  local measurements_table={}
  for i=3,12,2 do
    table.insert(measurements_table, data_table[i])
  end

-- report data to Mission Planner, not necessary all the time
  gcs:send_text(7, "pres:" .. string.format(" %.2f \r\n",
     measurements_table[1]) ..
                  "temp1:" .. string.format(" %.2f \r\n",
                     measurements_table[2]) ..
                  "temp2:" .. string.format(" %.2f \r\n",
                     measurements_table[3]) ..
                  "hum:" .. string.format(" %.2f \r\n",
                     measurements_table[4]) ..
                  "temp3:" .. string.format(" %.2f",
                     measurements_table[5])
  )

  -- return whether data input data matched needed format (table with 5
  --elements)
  return log_data(measurements_table)
```

```
34
35 end
```

We first take the message string and perform a regular expression (regex or regexp) match on the string. Here the Lua regex '%$(.*)%*' first looks for a '$', once it finds one, it then matches any characters after the '$' up until it finds a '*'. This extracts the main body of the message, which contains the data that we need to process for logging. An example of a full message can be seen below.

```
1 $UKPTH,000E,098152.5,Pa,23.17,C,22.90,C,42.21,%,22.45,C*4A<CR><LF>
```

Before we continue, we check if the regex failed, if it has, it will have returned a value of `nil`. We check for this, and if this is true, we return false for the caller to handle.

After we match the main body of the message, we can start preparing to extract the date from it. First we initialize the `data_table` table, which is where we will store each of the messages sections for processing.

Next we perform another regex on the message body. Here the regex '([^",".. "]+)', or more simply written as '([^,]+)' takes the string, and matches every character up until it finds a ','. It does this for all of the segments in the string. We then use the for loop to iterate over all of these segments and place them into are previously defined `data_table`.

We then check the size of `data_table` to ensure it got all twelve segments we are expecting, if not, we return false for the caller to handle.

We then take the measurement values from `data_table` and place them into a new table called `measurements_table`.

After we have extracted the measurement values, we can optionally send the values to the Mission Planner output, but is not required for the script to function.

Finally we call `log_data` and pass in the `measurements_table` as an argument. `log_data()` returns true or false depending on whether the table is the correct size. We then return this boolean value to the caller for them to handle.

For information on Lua's regular expressions, you can view these pages:

- Pattern-Matching Functions
- Lua pattern matching

For making patters and regular expressions in Lua, you can use these web tools:

- Lua Patterns Viewer
- Lua Pattern Tester

Logging

Logging Data

Snippet of the `log_data()` function, with comments removed.

```
function log_data(measurements_table)
  if #measurements_table ~= 5 then
    return false
  end
  logger:write('SAMA', 'pres,temp1,temp2,hum,temp3,error',
               'NNNNNN',
               'POO%O-',
               '------',
               measurements_table[1],
               measurements_table[2],
               measurements_table[3],
               measurements_table[4],
               measurements_table[5],
               "Normal")
  return true
end
```

Above is the `log_data()` function. This function takes in a table as an argument.

The function first checks if the table that is passed to it is the correct size, in this case 5, as that is the number of sensors on the PTH. If it does not pass this check, `log_data()` returns a **`false`** value, which is processed by the caller.

If the table passes this check, we then write the data to the BIN file.

The `logger:write` method take several arguments to define the various parameters that go into the log file.

The first argument, `'SAMA'`, is the section name for the data we are going to log in the file. This name has to be at most 4 characters, and cannot be the same as any other section name that ArduPilot logs. The second argument, `'pres,temp1,temp2,hum,temp3,error'`, specifies the name of each piece of data logged. These labels are stored under the section name in the log file, in total these names cannot exceed 64 characters.

The third argument, `'NNNNNN'`, specifies the type of each label. In this case 'N', specifies a **`char`**`[16]`, which is a string of a maximum of 16 characters. The fourth and fifth arguments specify the units and the multiplier of each of the units respectively. In the fourth argument 'P' represents Pascals, for the pressure measurement, 'O' represents degrees Celsius, for the temperature measurements, and '%' for percentage, for the humidity measurements, and '-' for no units/string for the error column. For the fifth argument, the '-' specifies that we want no multiplier applies to our data.

Further explanations on the format, unit, and multiplier types can be found here.

Once we specify the parameters for the data that is going to be logged, we then pass in the data we would like to log in the file. In this case, we use the 5 elements in the `measurements_table` table, and the string `"Normal"` for the error column. These are in the same order as the labels we specified in the second argument.

Further explanation on the arguments of the `logger:write()` method can be found here.

Once we log the data we simply return true to the caller for them to handle. `logger:write()` unfortunately does not return a value to tell us whether it was successful so we can only assume that it wrote to the BIN file correctly.

Logging Errors

The `log_error()` function, with comments removed.

```
function log_error(error_type)
  logger:write('SAMA', 'pres,temp1,temp2,hum,temp3,error',
               'NNNNNN',
               'POO%O-',
               '------',
               '0', '0', '0', '0', '0', error_type)
end
```

The `log_errors()` is very similar to the `log_data()` function, the only difference is that instead of writing any specific data, we simply write zeros to the log file, and log the type of error as a string (which will originate from the `error_list` table). We do this as it is very obvious in the log file when there is an error, and we deal with it easily during post processing.

For an explanation of the arguments, in `logger:write()`, you can look in the "Logging Data" section

Important Notes

If a Lua script has an error that the Lua interpreter detects, the script it generally not able to be restarted until the autopilot is manually restarted or a restart script command is sent. This is why there are several checks to ensure that the data parsing operations work as expected.

Doing this, and letting the script still run after an error is detected is important, as it prevent the script from crashing from a minor issue that fixes itself immediately, such as minor data corruption, or a short in a sensor connection, that causes a temporary disconnection. After the issue resolves itself, assuming the script can handle the error, the script can continue logging without having to land a drone and restart the script, which is vital for long and important flights.

Important Links

Below is a list of the URLs linked to in the document in case that the hyperlinks are not useable or reachable, such as if the document is printed on paper.

1. ArduPilot Serial Dump Example
    - https://github.com/ArduPilot/ardupilot/blob/master/libraries/AP_Scripting/examples/Serial_Dump.lua
2. Exclusive OR Wikipedia Article
    - https://en.wikipedia.org/wiki/Exclusive_or
3. Formatting, Units, and Multipliers in ArduPilots Logging System
    - https://github.com/ArduPilot/ardupilot/blob/master/libraries/AP_Logger/README.md
4. ArduPilot adding a new log message
    - https://ardupilot.org/dev/docs/code-overview-adding-a-new-log-message.html
5. Lua Pattern-Matching Functions
    - https://www.lua.org/pil/20.1.html
6. Lua Pattern Matching
    - https://riptutorial.com/lua/example/20315/lua-pattern-matching
7. Lua Patterns Viewer
    - https://gitspartv.github.io/lua-patterns/
8. Lua Pattern Tester
    - https://montymahato.github.io/lua-pattern-tester/