## Table of Contents

## 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.

### Serial Initialization

```lua
-- initialize serial connection
local BAUD_RATE = 9600

-- find the serial first (0) scripting serial port instance
-- SERIALx_PROTOCOL 28
local PORT = assert(serial:find_serial(0),"Could not find Scripting
    Serial Port")

-- begin the serial port
PORT:begin(BAUD_RATE)
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.

```lua
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.

```lua
function update()
  local n_bytes = PORT:available()

  -- If there is one or more loops that are unsuccessful, send an error
     message
  -- to Mission Planner and write all zeros to the log file
  if n_bytes <= 0 then
    log_error()
    gcs:send_text(0, "ERROR: PTH has failed to send data")
    -- write zeros to BIN file to make it clear that the sensor is
       disconnected
  end

  while n_bytes > 0 do
    -- only read a max of 515 bytes in a go
    -- this limits memory consumption
    local buffer = {} -- table to buffer data
    local bytes_target = n_bytes - math.min(n_bytes, 512)
    while n_bytes > bytes_target do
      table.insert(buffer,PORT:read())
      n_bytes = n_bytes - 1
    end
```

We first get the amount of available bytes from our scripting serial port instance and store it in `n_bytes`. We then check the value of `n_bytes` to make sure it is not zero. If it is zero, this means that the sensor did not send any data over the serial connection. This indicates that the sensor is either disconnected in some way, or we have read the serial line before the sensor has sent another message (This is unlikely since we schedule update to run at the same time the sensor should send its data.).

If this is the case, we then write all zeros to the log file with the `log_error()` function (This will be discussed further in the "Logging" section).

Once we log the error to the BIN file, we then send an error message to the Mission Planner output with the following line.

```
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.

If we do pass this check, meaning the sensor did send data, we can then start getting the data from the serial line. We start by checking to see how many bytes are available, we then set the amount of times we want to read off of the serial line, up to a max of 515 bytes. 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.

Second 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()
        gcs:send_text(0, "ERROR: PTH data was not successfully parsed
            or not written to BIN file correctly!")
        gcs:send_text(0, "Incoming string: " .. data .. string.format("
             size: %d", #data))
      end
    else
      log_error()
      gcs:send_text(0, "ERROR: PTH Data failed checksum, check sensor!"
          )
      gcs:send_text(0, "Incoming string: " .. data .. string.format("
          size: %d", #data))
    end
  end

  return update, 1000 -- reschedules the loop every 1000ms (1 second,
      max since sensor only sends 1 message every second)
end
```

Once we have placed the message from the serial line into out `buffer` table. We can now start processing the data. First 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.

```
$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.

```
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" section.

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.

```lua
if not (parse_data(data)) then
  log_error()
  gcs:send_text(0, "ERROR: PTH data was not successfully parsed or not written to BIN file correctly!")
  gcs:send_text(0, "Incoming string: " .. data .. string.format(" size: %d", #data))
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.

```lua
return update, 1000 -- reschedules the loop every 1000ms (1 second, max since sensor only sends 1 message every second)
```

The above statement returns from the `update()` and calls the function again. The second argument in the return statement specifies the number of milliseconds the autopilot should wait before running the `update()` function again. In this case, 1000 milliseconds (1 second) was chosen. This is because the PTH sensor only reports data every second, meaning running the script faster than that would cause us to read an empty serial line, causing an error.

After 1 second, the `update()` function is then run again, continuing the process of logging the data to the BIN file.

## Parsing Data

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

```lua
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

  return log_data(measurements_table)

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

end
```

### Verifying Checksum

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

```lua
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
```

## 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',
               'NNNNN', 'POO%O', '-----',
               measurements_table[1],
               measurements_table[2],
               measurements_table[3],
               measurements_table[4],
               measurements_table[5])
  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'`, 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, `'NNNNN'`, specifies the type of each label. In this case '`N`', specifices 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. For the fifth argument, the '−' specifies that we want no multiplier applies to our data.

Further explanations on the various 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, which is 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.

```lua
function log_error()
    logger:write('SAMA', 'pres,temp1,temp2,hum,temp3',
                        'NNNNN',
                        'POO%O',
                        '-----',
                        '0', '0', '0', '0', '0')
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. 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. 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