Table of Contents

Hardware Setup

The sonic anemometer, specifically the LCJ Capteurs CV7-OEM, uses the RS232 serial standard. This means to use the sonic anemometer with the autopilot, a RS232 to standard UART Serial adapter is required.

In Mission Planner, with the autopilot connected. Go to the "CONFIG" tab, and select the "Full Parameter List". In the parameter list, you will need to find one of the "`SERIAL#`" sections. This will depend on what autopilot you are using and what port you are connecting the anemometer to. Refer to the documentation of your specific autopilot for the mapping of the physical hardware ports, to their name in Mission Planner.

Under the "`SERIAL#`" section, change the "`SERAIL#_BAUD`" parameter to "`4`", or in the options drop-down, select "`4800`", to set the baud rate to 4800.

Then under the "`SERIAL#_PROTOCOL`" section, change the parameter to "`28`" or in the options drop-down, select "`Scripting`", to enable scripting on that serial port. This allows for the Lua script to access this port.

It is important to note that if you have multiple scripting ports active at once, the script will by default select the first port. So to ensure that the script is connected to the correct port, you can edit the below line:

```
local PORT = assert(serial:find_serial(0),"Could not find Scripting Serial Port")
```

To change `serial:find_serial(0)`, which finds the first serial port, to `serial:find_serial(1)`, to find the second serial port.

Instead of editing the script, you can simply ensure that the anemometer is on the first scripting port, and the other serial scripting device is set up for the second serial port.

Error Messages Overview

Quick Reference Table

Mission Planner Output	Log File	Error Type
ERROR ANEM: Disconnected Sensor	No data received	Disconnected sensor
ERROR ANEM: Invalid message frame	Invalid frame	Corrupted/Incomplete data
ERROR ANEM: Data failed checksum	Checksum fail	Corrupted/Incomplete data
ERROR ANEM: Failed to parse data	Parsing fail	Corrupted/Incomplete data

Explanation

Disconnected Sensor

If the script stops receiving data from the anemometer, it will send the following error to the Mission Planner output.

```
6/11/2024 11:02:42 AM : ERROR ANEM: Disconnected Sensor
```

The error message will also appear in the log file, under the "ANEM" section in the errors column, with the message "No data received".

A disconnected sensor error can occur when there is an improper connection between either the anemometer and the RS232 to UART converter, or from the RS232 converter to the autopilot.

Corrupted or Incomplete data

If the script receives corrupted data, or an incomplete message, there are several errors that can appear.

1. Invalid Frame

   An invalid frame error is when the script detects that the message it has received from the serial line does not contain the standard NMEA-0183 sentence starting or ending characters, being '$' and <CR><LF> respectively. If the script detects this, it will send the following error to the Mission Planner output.

   ```
   6/11/2024 11:02:42 AM : ERROR ANEM: Invalid message frame
   ```

   The error message will also appear in the log file, under the "ANEM" section in the errors column, with the message "Invalid frame".

   An invalid frame error can occur when the data coming into the autopilot has been corrupted. Likely due to noise on the serial line, or an improper connection between either the anemometer and the RS232 to UART converter, or from the RS232 converter to the autopilot.

   An invalid frame can also occur when the script reads the parts of two messages, and tries to decode it as a single message. This can happen if there are multiple messages in the serial queue at once. This is usually happens when the anemometer starts sending data before the script can fully initialize and start decoding messages, or when the script running to slow and cannot keep up with the amount of messages that are being sent. The script is designed to handle when this happens and will clear the queue to ensure it can catch back up or to 're-align' the messages in the queue.

2. Checksum Fail

   A checksum fail error is when the script detects that the message has been corrupted in some manner. It does this by verifying that the checksum that is sent with the message matches with the checksum the script calculates.

   If the checksums do not match, or if there is an issue when extracting the checksum from the message, the script will send the following message to the Mission Planner output.

   ```
   6/11/2024 11:02:42 AM : ERROR ANEM: Data failed checksum
   ```

   The error message will also appear in the log file, under the "ANEM" section in the errors column, with the message "Checksum fail".

   An checksum fail error will occur when the data coming into the autopilot has been corrupted. Likely due to noise on the serial line, or an improper connection between either the anemometer

and the RS232 to UART converter, or from the RS232 converter to the autopilot.

3. Parsing Fail

   A parsing fail error is when the script cannot properly extract the data from the message.

   If the script is unable to parse the message, it will send the following error message to the Mission Planner output.

```
6/11/2024 11:02:42 AM : ERROR ANEM: Failed to parse data
```

   The error message will also appear in the log file, under the "ANEM" section in the errors column, with the message "Parsing fail".

   While possible, it is unlikely that a parsing error will come from a corrupted or incomplete message, since the message frame verification and checksum verification will catch the majority of the corrupted or incomplete messages. It is more likely that `parse_data()` has been edited and there is a bug with either the regexes or with how the function performs data extraction.

Code Overview

> **Note**: The comments have been removed from many of the code snippets for clarity and brevity

Global Variables and Constants

```lua
local BAUD_RATE = 4800

local MAX_MESSAGE_LENGTH = 31

local SCHEDULE_RATE = 100 --milliseconds
local TIME_BETWEEN_DATA = 512 --milliseconds
assert((SCHEDULE_RATE < TIME_BETWEEN_DATA), "ANEM Loop reschedule rate
    to long")
local LOOPS_TO_FAIL = (TIME_BETWEEN_DATA // SCHEDULE_RATE) + (1)

local ERROR_LIST = {
  no_data       = "No data received",
  checksum      = "Checksum fail",
  parsing       = "Parsing fail",
  invalid_frame = "Invalid frame",
}

local MESSAGE_INFO = {
  ["IIMWV"] = {
    length = 28,
    fields = 6,
```

```
    measurements = 2
  },
}

-- Omitted: Serial Initialization

local loops_since_data_received = 0
```

We start by initializing all of our global constants and variables. Note that all constants are in all caps, while global variables are all lower case like the other variables present in the script.

`BAUD_RATE` defines the baud rate of the anemometer, in this case it is 4800. This is used during the serial initialization.

`MAX_MESSAGE_LENGTH` defines the maximum length (in bytes/characters) of the messages we will receive from the anemometer. We use this in the `update()` function when we read data from the serial line.

`SCHEDULE_RATE` defines how long do we want to wait in milliseconds before rerunning the `update()` function.

`TIME_BETWEEN_DATA` defines the time between data transmissions from the sensor, in milliseconds. We use this and the `SCHEDULE_RATE` number to calculate the `LOOPS_TO_FAIL` value. We use this when determining if the sensor is disconnected. We also verify that `SCHEDULE_RATE` is less than `TIME_BETWEEN_DATA` to ensure that the script is rescheduled fast enough to ensure that it can keep up with the flow of messages from the anemometer.

The `ERROR_LIST` table holds a set of key-value pairs which correspond to the different kinds of errors that can be experienced. These are referenced when using the `log_error()` function to pass in the string we would like to log for the error message. **Note**: all of the string in the `ERROR_LIST` table must be 16 character (bytes) or less to be properly logged.

The `MESSAGE_INFO` table holds a set of key-value pairs which map the NMEA message headers that we expect and would like to decode to a table of information about the message. Specifically it holds the length of the message, the number of fields that are in the message, and the number of measurements that are in the message. This table gets referenced when parsing the data from the message to ensure that the data was successfully extracted.

The `loops_since_data_received` variable holds the amount of loops we have gone through since we have received any new data. This is incremented once every loop without data, and is compared with the `LOOPS_TO_FAIL` value before we log an error.

Serial Initialization

```
local PORT = assert(serial:find_serial(0),"Could not find Scripting
    Serial Port")
PORT:begin(BAUD_RATE)
PORT:set_flow_control(0)
```

To initialize the serial connection, we call the `find_serial()` method, which will find a scripting serial port instance. In the code snippet above, we call `serial:find_serial(0)`, which will look for the first instance of a scripting enabled serial port. If you have multiple scripting serial ports, you can change the argument of the `find_serial()` from a `0` to a `1`, which will look for the second scripting enables serial port, as opposed to the first.

If we do not find a serial port that has scripting enabled, we throw an error to alert the user that the autopilot has not been properly set up to use the sensor.

After finding the serial port, we store the interface in the `PORT` variable. To start the connection, we call the `begin()` method and pass in the `BAUD_RATE` constant that we defined earlier. We also disable UART flow control with the `set_flow_control()` method, since the anemometer does not support it.

Update Function

Once the constants are defined and 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 statements.

```
--clear the queue to prevent message build up before we schedule the
    loop
PORT:readstring(PORT:available():toint())
return update() -- run immediately before starting to reschedule
```

During the initialization process, we can collect one or more messages in the serial queue, which can cause issues when we start trying to read messages from the serial line. To prevent any errors that can arise from having more that one message in the queue, we simply clear it before we schedule the update function.

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

  if n_bytes <= 0 then
    loops_since_data_received = loops_since_data_received + 1
    if loops_since_data_received >= LOOPS_TO_FAIL then
      log_error(ERROR_LIST.no_data)
      gcs:send_text(0, "ERROR ANEM: Disconnected Sensor")
    end
    return update, SCHEDULE_RATE
```

```
11    end
```

For every iteration we first start by checking how many bytes are available in the serial queue by using the `available()` method for the serial interface. We then check if that value is zero. If it is this means that the anemometer has not sent any data yet. We then add one to the `loops_since_data_received` variable, and check if the value is over the `LOOPS_TO_FAIL` limit. If it is, we then log an error to the log file, and send an error message to Mission Planner.

```
local message_string = PORT:readstring(MAX_MESSAGE_LENGTH)

if (message_string == nil or #message_string <= 0) then
  return update, SCHEDULE_RATE
end

loops_since_data_received = 0

if not (verify_valid_frame(message_string)) then
  PORT:readstring(PORT:available():toint())
  log_error(ERROR_LIST.invalid_frame)
  gcs:send_text(0, "ERROR ANEM: Invalid message frame")
  gcs:send_text(7, message_string)
  return update, SCHEDULE_RATE
end
```

If we have received data from the anemometer, we call the `readstring()` method on the serial interface to get a string that is at most `MAX_MESSAGE_LENGTH` bytes long.

To prevent a situation where `readstring()` returns `nil` value or an empty string and we try to parse the data, we simply check for both of these conditions, and reschedule the function if either of them are true.

> Instead of reading each byte individually off of the serial line, and checking to see if we have reached the end of a NMEA message, we simply pull the max message length that we are expecting from the sensor off the serial line each time.
>
> If we are parsing and logging the data fast enough, and are rescheduling the loop often enough, we will only have one message in the queue at a time. Which means even if the current message in the queue is shorter than the maximum message length, we will not pull parts of another message, since there is no second message in the queue

After we have passed the first initial checks to make sure we have received a message, we then can call the `verify_valid_frame()` function, which will take in the message string, and check for the NMEA-0183 sentence start and ending characters. Which are "`$`" and `<CR><LF>` respectively. If the function does not find both of these, it will return false. If the function does return false, we know that either we have a corrupt or incomplete message, or we have read the parts of two separate

messages.

We first clear the serial queue by reading the rest of available bytes into a string, and doing nothing with them. This will "re-align" the serial queue to ensure that the first byte we read will be the start of the sentence, and that there are no messages in the queue to ensure we do not read a part of the second message.

We then log an error to the autopilot's log file with the `log_error()` method. Which takes a 16 byte string as its input, which we pull from the `ERROR_LIST`, using the `invalid_frame` key to pass in the desired string. We also send an error to the Mission Planner output, specifying what sensor is having the error, and what king of error we are experiencing. We then return and reschedule the update function to run again in `SCHEDULE_RATE` milliseconds.

```
1  if not (is_message_useful(message_string)) then
2    return update, SCHEDULE_RATE
3  end
4
5  if not (verify_checksum(message_string)) then
6    log_error(ERROR_LIST.checksum)
7    gcs:send_text(0, "ERROR ANEM: Data failed checksum")
8    gcs:send_text(7, message_string)
9    return update, SCHEDULE_RATE
10 end
```

If we have received a valid frame from the anemometer, we can now check if the message is one that holds data we would like to log. We pass the message string into the `is_message_useful()` function, which extracts the NMEA message header, and checks if its one of the messages we would like to parse. In our case, we only would like to parse the "IIMWV" message, which contains wind direction and wind speed data.

If it is not the message we would like to parse we simply return and wait to process the next message. If it is the "IIMWV" message, we can continue on to the checksum verification.

Here I will give general explanation on how the checksum is calculated and verified. The specifics of how the `verify_checksum()` works will be discussed in the "Verifying Checksum" section.

Below is an example of one of the messages that the anemometer can send, which is in the NMEA-0183 message format standard. 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  $IIMWV,179.0,R,000.3,M,A*32<CR><LF>
```

To calculate the checksum from the message, we first 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.

```
IIMWV,179.0,R,000.3,M,A*32
```

Once we have the "main body" of the message, we can now calculate the checksum. For the NMEA-0183 standard, we calculate the checksum by performing a bitwise exclusive OR (XOR) of all of the characters that are in the "main message", not including the "`$`" and "`*`".

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

In the example above, we start out with the ASCII character `I`. We then XOR `I` with the next character in the string, which happens to be another `I`. 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 `M`. 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, call `log_error()` to log an error in the BIN file, and return and reschedule the `update()` function.

```
if not (parse_data(message_string)) then
  log_error(ERROR_LIST.parsing)
  gcs:send_text(0, "ERROR ANEM: Failed to parse data")
  gcs:send_text(7, message_string)
  return update, SCHEDULE_RATE
end
```

Now that we have passed all of the checks, we can now begin to parse and log the message.

Again, I will be giving a general overview of how parsing and logging works, while more specific explanations will be in the "Logging" and "Parsing Data" sections.

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

After we have all of the fields in a table, we then iterate over the table, looking at each of the fields to see if they contain a floating point number. If they are, we store them in a separate measurements table.

Once we have the measurements from the message, we check what message we are handling, and verify that we parsed all of the expected measurements for this message. We then send the measurements to the appropriate function. In this case we only have one message we handle, so there is only one logging function.

In the logging function, we define the measurement section that will appear in the log file, which is "ANEM" in this case. We also define all of the data fields that we would like to record. In the case of the "IIMWV" message, we have "angle", "speed", and "error". We then write this data to the log file.

If there were any errors in extracting the fields or the measurements from the message, or if there was an issue in the logging process, we log an error in the log file, and send an error to the Mission Planner output.

```
  return update, SCHEDULE_RATE
end
```

After we have logged the data into the log file, we can now reschedule the `update()` function to run again in `SCHEDULE_RATE` milliseconds. After that amount of time, the loop will run again, processing the next message.

Verifying Valid Frame

```
function verify_valid_frame(message_string)
  local head = string.sub(message_string, 1, 1)
  local tail = string.sub(message_string, #message_string-1, #
      message_string)

  if (head == "$") and (tail == "\r\n") then
    return true
  end
  return false
end
```

To verify that we have a valid NMEA-0183 message frame, we need to check if we have the standard sentence starting and ending characters for NMEA-0183 messages, being "`$`" and "`<CR><LF>`" respectively.

To get these characters, we simply use the `string.sub()` method to extract the sub-strings that contains the sentence delimiters. In `strings.sub()`, we specify the string we are going to be working on, and then we pass in the starting and ending indices of the sub-string we want (The sub-string indices are inclusive). For the "head", we simply take the first character in the string. For the "tail", we specify the last two indices of the string, which if the message is valid, will contain the two ending characters.

We then take the sub-strings and verify that they do contain the starting and ending characters. If both sets are present, we return true to the caller. If one or both of the sub-strings do not match their respective sentence delimiters, we return false.

Is Message Useful

```lua
function is_message_useful(message_string)
  local message_type = message_string:match("%$(.-),")

  if message_type == nil then
    return false
  elseif message_type == "IIMWV" then
    return true
  end
  return false
end
```

Since the anemometer sends several messages that we do not care about, it is important that we can throw out any messages early into the loop to prevent wasted time.

We first use the below regex:

```
%$(.-),
```

Which will first find the "$" character, which signifies the start of a NMEA-0183 message. It then matches any character up until the first comma. This field, according to the NMEA-0183 standard, hold the message identifier.

We first check and see if the regex was successful by making sure the `message_type` variable is not `nil`. If we did in fact get a message header we then check if it is the "IIMWV" message header, which is the only message we would like to parse. If it is, we return true, if it is not, we return false.

Verifying Checksum

```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

  return (checksum == incoming_checksum)
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.

```
$IIMWV,179.0,R,000.3,M,A*32
```

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 capital 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 extracted 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 patterns and regular expressions in Lua, you can use these web tools:

- Lua Patterns Viewer
- Lua Pattern Tester

Parsing Data

```lua
function parse_data(message_string)
  local data_string = message_string:match("%$(.*)%*")
  if data_string == nil then
    return false
  end

  local message_type = message_string:match("%$(.-),")
  if message_type == 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 ~= MESSAGE_INFO[message_type].fields then
    return false
  end

  local measurement_table={}
  for i = 1, #data_table do
    local m = string.match(data_table[i], "%d*%.%d*")
    if m ~= nil then
      table.insert(measurement_table, m)
    end
  end
```

```
28
29    if #measurement_table ~= MESSAGE_INFO[message_type].measurements then
30      return false
31    end
32
33    if message_type == "IIMWV" then
34      return log_wind_speed(measurement_table)
35    end
36
37     return false
38  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  $IIMWV,179.0,R,000.3,M,A*32
```

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 extract the message header with the regex "%$(.-),", which will extract the first field in the message, we will use this later in the function. We also verify that regex worked by checking if the value is `nil`.

After we match the main body of the message, we can start preparing to extract the data 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`, and make sure it matches up with the number of fields that we know are present in the message we are parsing. In our case the "IIMWV" message has six fields. If we find that this is not true, we return false for the caller to handle.

We then iterate over `data_table` checking each value and seeing if it matches with the regex "%d*%.%d*", which accepts any floating point number. We then place any matches

Finally we call our log function and pass in the `measurements_table` as an argument. The log function 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 patterns and regular expressions in Lua, you can use these web tools:

- Lua Patterns Viewer
- Lua Pattern Tester

Logging

Logging Data

```lua
function log_wind_speed(measurement_table)
  if #measurement_table ~= MESSAGE_INFO["IIMWV"].measurements then
    return false

  logger:write('ANEM', 'angle,speed,error',
               'NNN',
               measurement_table[1],
               measurement_table[2],
               'Normal')
  return true
end
```

The `log_wind_speed()` is the function that logs the data that comes from the "IIMWV" message from the anemometer, in the form of a table.

The function first checks if the table has the correct number of measurements by referencing the measurements value of the "IIMWV" message in the `MESSAGE_INFO` table. If the values differ, we return false to the caller. If they are the same, we can continue to log the values.

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

The first argument, '`ANEM`', 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, '`angle,speed,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, '`NNN`', specifies the type of each label. In this case '`N`', specifies a **`char`**`[16]`, which is a string of a maximum of 16 characters.

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 2 measurements 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

```
function log_error(error_type)
  logger:write('ANEM', 'angle,speed,error',
               'NNN',
               '0',
               '0',
               error_type)
end
```

The `log_error()` is very similar to the data logging 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 usable or reachable, such as if the document is printed on paper.

1. Exclusive OR Wikipedia Article
    - https://en.wikipedia.org/wiki/Exclusive_or
2. Formatting, Units, and Multipliers in ArduPilot's Logging System
    - https://github.com/ArduPilot/ardupilot/blob/master/libraries/AP_Logger/README.md
3. ArduPilot adding a new log message
    - https://ardupilot.org/dev/docs/code-overview-adding-a-new-log-message.html
4. Lua Pattern-Matching Functions
    - https://www.lua.org/pil/20.1.html
5. Lua Pattern Matching
    - https://riptutorial.com/lua/example/20315/lua-pattern-matching
6. Lua Patterns Viewer
    - https://gitspartv.github.io/lua-patterns/
7. Lua Pattern Tester
    - https://montymahato.github.io/lua-pattern-tester/
8. Supported log file data types
    - https://github.com/ArduPilot/ardupilot/blob/master/libraries/AP_Logger/README.md
9. `logger:write()` method documentation
    - https://ardupilot.org/dev/docs/code-overview-adding-a-new-log-message.html