Final Internship Report, Part 2 - Code Documentation

Author: Tan Xuan De, Kelvin

```
- kelvin.xuande@u.nus.edu, kelvinxuande@gmail.com
```

This is the second part of a 2-part report which seeks to address the following:

- Source code(s) of project, their explanations and logic flow
- Thought processes that went into using the components in the source code(s)

Appendix A:

- Validity-check for function to group bytes into complete message (in list form) and to keep remainder for next cycle
- Script profiling
- Changelog - version3 to version4/ version4_dev

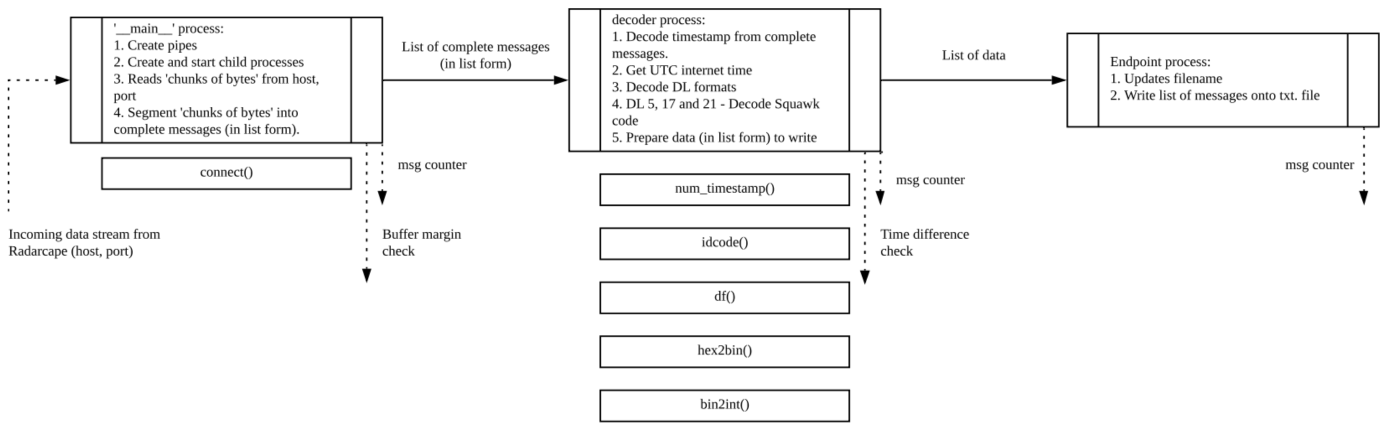
Appendix B: Future work

- Debug timestamp decoding

Credits and Acknowledgements:
The Author would like to extend his sincere thanks to the people who contributed to the project, including but not exhaustively:

- junzis/pyModeS (https://github.com/junzis/pyModeS) for the source code on pyModeS and accompanying materials for reference.
- Oliver Jowett (https://github.com/mutability) for his invaluable time, advice and assistance rendered.
- Project supervisors for the crucial guidance and required hardware.

Code Architecture:


'__main__' process:
1. Create pipes
2. Create and start child processes
3. Reads 'chunks of bytes' from host, port
4. Segment 'chunks of bytes' into complete messages (in list form).
connect()
Incoming data stream from Radarcape (host, port)
msg counter
Buffer margin check
List of complete messages (in list form)
decoder process:
1. Decode timestamp from complete messages.
2. Get UTC internet time
3. Decode DL formats
4. DL 5, 17 and 21 - Decode Squawk code
5. Prepare data (in list form) to write
num_timestamp()
idcode()
df()
hex2bin()
bin2int()
msg counter
Time difference check
List of data
Endpoint process:
1. Updates filename
2. Write list of messages onto txt. file
msg counter

- Since the architecture of the code resembles a 'pipeline', the flow of this document would follow the order in which actual messages are processed from start to end; as opposed to following strictly the line numbers in the actual code.
- Similar to this example, any code snippets or diagrams would appear before their explanations.

In [9]:

```python
"""Import required libraries"""
import socket
import time
from datetime import datetime
import multiprocessing
```

We start the script with importing the necessary python libraries that we will be using.
Here is a brief breakdown of the libraries:

- socket (https://docs.python.org/3.7/library/socket.html) - to create and handle sockets.
- time (https://docs.python.org/3/library/time.html) - provides various time-related functions.
- datetime (https://docs.python.org/3/library/datetime.html#module-datetime) - supplies classes for manipulating dates and times in both simple and complex ways.
- multiprocessing (https://docs.python.org/3/library/multiprocessing.html) - package that supports spawning processes using an API similar to the threading module.

In [ ]:

```python
"""Main process that sets up architecture and collect data to pipe"""
if __name__ == '__main__':

    # Create pipes:
    decode_start, parent_end = multiprocessing.Pipe(False)
    endpoint_start, decode_end = multiprocessing.Pipe(False)

    # Create new child processes and give them pipe 'starts' and 'ends':
    decoding_process = multiprocessing.Process(target=decoder, args=(decode_start, dec
    endpoint_process = multiprocessing.Process(target=endpoint, args=(endpoint_start,)

    # Start both child processes:
    decoding_process.start()
    endpoint_process.start()

    # Establish connection:
    time_now = str(datetime.utcnow())
    print("**************")
    print("%s | Script initiated" %(time_now))
    sock = connect()
```

Main process

We then move onto the start of the pipe, the 'main' function (and the main process, in this case), sets up the 'pipeline' by doing the following:

- Creating two pipes (https://docs.python.org/3.7/library/multiprocessing.html#pipes-and-queues)
- Creating two 'child' processes (https://docs.python.org/3.7/library/multiprocessing.html)
- Calling the socket function

Key points:

Multiprocessing, threading and asyncio (https://realpython.com/python-concurrency/):
The default Python interpreter was designed with simplicity in mind and has a thread-safe mechanism - the so-called “GIL” (Global Interpreter Lock). In order to prevent conflicts between threads, it executes only one statement at a time (so-called serial processing, or single-threading).
With multiprocessing, Python creates new processes. A process here can be thought of as almost a completely different program, though technically they’re usually defined as a collection of resources where the resources include memory, file handles among other things. One way to think about it is that each process runs in its own Python interpreter. Because they are different processes, a multiprocessing program can run simultaneously on different cores.

When to use which? (https://realpython.com/python-concurrency/):

- I/O-bound problems cause your program to slow down because it frequently must wait for input/output (I/O) from some external resource. They arise frequently when your program is working with things that are much slower than your CPU. In scenarios such as these, it would be more beneficial to use Threading.
- On a CPU-bound problem, however, there is no waiting. The CPU is cranking away as fast as it can to finish the problem. In Python, both threads and tasks run on the same CPU in the same process. That means that the one CPU is doing all of the work of the non-concurrent code plus the extra work of setting up threads or tasks. In cases such as these, it would be more beneficial to use Multiprocessing.

Pool vs Process in Multiprocessing (https://www.ellicium.com/python-multiprocessing-pool-process/):

- The pool distributes the tasks to the available processors using a FIFO scheduling. It works like a map reduce architecture. It maps the input to the different processors and collects the output from all the processors. After the execution of code, it returns the output in form of a list or array. It waits for all the tasks to finish and then returns the output. The processes in execution are stored in memory and other non-executing processes are stored out of memory.
- The process class puts all the processes in memory and schedules execution using FIFO policy. When the process is suspended, it pre-empts and schedules new process for execution.
- Pool allows you to do multiple jobs per process, which may make it easier to parallelize your program. If you have a million tasks to execute in parallel, you can create a Pool with number of processes as many as CPU cores and then pass the list of the million tasks to pool.map. The pool will distribute those tasks to the worker processes(typically same in number as available cores) and collects the return values in the form of list and pass it to the parent process. Launching separate million processes would be much less practical (it would probably break your OS).
- On the other hand, if you have a small number of tasks to execute in parallel, and you only need each task done once, it may be perfectly reasonable to use a separate multiprocessing.process for each task, rather than setting up a Pool.

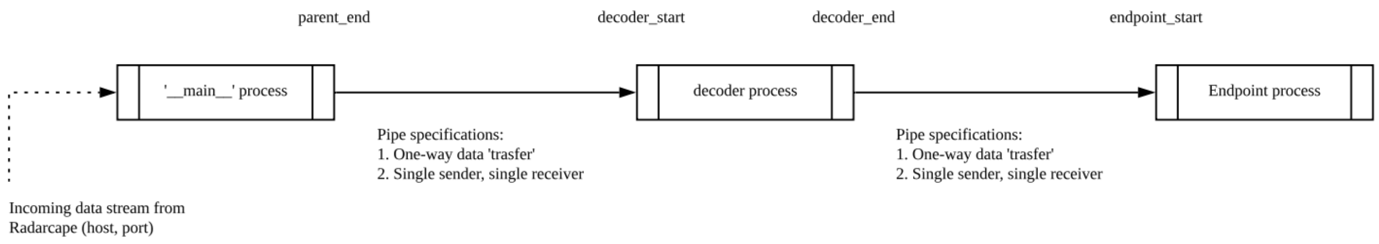
For our application:
It was hypothesized at the start of the project that the script process is CPU-bound. i.e. processing the incoming data is slower as compared to collecting incoming data. It was hence a concern that if a script with a single process is used, incoming data might be lost since the script would spend most of the time processing the incoming data rather than collecting new messages. *Hence, it was desirable to use multiprocessing.*
Considering our use case deeper, since each message need only be decoded once and managing a single process was much easier in terms of both creation and debugging, it was easier to use multiprocessing.process.
When the decoding process is guaranteed to be stable in the future, using a pool of workers and having each worker decode some (complete) messages in a list and returning the results in sequence; may be explored.
Additional Reference (http://blog.shenwei.me/python-multiprocessing-pool-difference-between-map-apply-map_async-apply_async/)

Limitation: Utilising multiprocessing still requires the 'overall processing rate' to be faster than the 'incoming data rate'. If this requirement is not met, the TCP buffer will inevitably run out of space/ memory to store unprocessed messages. A suitable analogy would be how water flows into a bucket faster than it is able to flow out.

[Pipes, Queues (and shared states) (https://docs.python.org/3.7/library/multiprocessing.html#pipes-and-queues)](https://docs.python.org/3.7/library/multiprocessing.html#pipes-and-queues): Since the processes are separate from one another, they do not share the same memory spaces, Hence, there needs to be a way to ‘pass’ data from one process to another. There are primarily three ways to achieve this – queue, pipe and shared states:

- For passing messages one can use Pipe() (for a connection between two processes) or a queue (which allows multiple producers and consumers).
- Queue: Appears to have more safeguards to prevent errors and also offer checks like queue.empty and queue.full. However, it is slower than using pipes since it was built on top of pipe.
- Pipe: Appears that more attention is required e.g. the need to ensure that pipe ‘starts’ and ‘ends’ are properly allocated. Although it seems to offer less checks (such as the two mentioned above), it is faster.
- Shared states: It was specially mentioned to avoid using this to shift large amounts of data between processes and that it is probably best to stick to using queues or pipes for communication between processes rather than using the lower level synchronization primitives. We will therefore avoid this method.

For our application:



- In order to prevent any potential errors, we have configured the pipe to be one-way instead of duplex, by setting duplex to be 'False'.
- In order to use pipes to do multiprocessing, we have to pass pipe 'starts' and/or 'ends' to the child processes to send data and/ or to receive data respectively. In this aspect, a pipe functions similar to a socket.

In [ ]:

```python
"""Function to connect to hardcoded host, port"""
def connect():
    host = "169.254.210.120"
    port = 10003
    while True:
        try:
            s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            # 10 seconds to return object before timeout:
            s.settimeout(10)
            s.connect((host, port))
            time_now = str(datetime.utcnow())
            print("%s | Server connected - %s:%s" %(time_now, host, port))
            print("%s | Collecting messages..." %(time_now))
            return s
        except socket.error as err:
            time_now = str(datetime.utcnow())
            print("%s | Socket connection error: %s. Reconnecting..." %(time_now, err)
            errorHandler('1a')
            time.sleep(3)
```

'Connect' Function - used only by the Main process

When this function is called by (and only by) the main function, it connects to a user-defined hardcoded host and port pair; and returns a socket object. This routine is typical and follows convention. For more details, consider reviewing the documentation for python (https://docs.python.org/3/library/socket.html) and linux (http://man7.org/linux/man-pages/man2/socket.2.html) sockets. In particular,

- SOCK_STREAM provides sequenced, reliable, two-way, connection-based byte streams. An out-of-band data transmission mechanism may be supported.
- An example of setting up sockets in python can be found here (https://realpython.com/python-sockets/).

More on TCP sockets:

- Loss of data (received, not processed in time and in the end overwritten) was initially a huge concern.
- Industrial expert Oliver Jowett confirmed TCP buffers to be lossless and that TCP data sitting in the TCP buffer (waiting to be retrieved and processed) is kept in FIFO (first-in, first-out) sequence. He argues that what we should be looking into is profiling and ensuring that the speed at each messages are processed can keep up with the rate at which messages are coming in.

To do this profiling,

- At the start and end of each child process, a counter was implemented. With this counter, we can quickly identify which process is creating a bottleneck in the entire pipeline.
- In addition, the code was written such that the decoded timestamp and the Internet UTC time is constantly written onto file and can be quickly compared. This would quickly allow us to spot any time delay abnormalies that would suggest that the decoding is not being carried out fast enough. This will be explored later.

In [ ]:

```python
# Initialise 'slow-changing' variables:
run_parent = True       # process control flag for key-board interrupt
min_margin = 16384      # variable for 'margin check', stores minimum margin
buffer = []             # buffer to store incoming data
main_message_count = 0  # keep count the number of complete messages
```

Next, we define some 'slow-changing' variables - variables that are not changed often.

In [ ]:

```python
# Start receiving data:
while run_parent:

    try:
        # Receive from socket:
        received = sock.recv(16384)

        # Check that we are receiving:
        if (len(received) == 0):
            errorHandler('1b')

        # Confirmed that we are receiving:
        else:

            # Check margin:
            margin = 16384 - len(received)

            # Check if margin is zero:
            if (margin == 0):
                errorHandler('1c')
                min_margin = margin

            # Even if not zero, update if there is a new minimum margin:
            elif (margin < min_margin):
                min_margin = margin

        # Extends list to add newly received elements:
        buffer.extend(received)

        # Initialise/ empty lists with each loop, after sending:
        completed_msg_list = []
        completed_msg = []
        i = 0
```

1. We start the loop to constantly retrieve data from our socket that we previously defined.
2. The variable 'run_parent' had been previously initialised to be set to True - and this sets the loop to run consistently.
3. We then try to receive data from the socket (with a buffer size (https://stackoverflow.com/questions/7174927/when-does-socket-recvrecv-size-return) we define) and thereafter check if we have indeed received something. When data has been read/ received from the the TCP buffer, it is 'consumed'.
4. We then carry out a margin check and update the 'minimum margin' if need be. The margin here refers to the subtraction of the size of the data coming in (i.e. amount of space used) from the size of the TCP buffer (space that is available), with each loop. This is to validate that we are at least fast enough in receiving the data from the socket, and that there is no data loss due to 'socket overflow'.

5. Once this is done, we then extend our buffer with the data that was received.

Since data comes in in the form of a data steam, a buffer may have many, one, parts of one, or even no messages.
We therefore need to group bytes together to form one complete message. According to Oliver and referencing source codes from pyModeS, this is done by looking out for 'flag bytes' and 'stuffed bytes'.

- '[Flag bytes (https://www.tutorialspoint.com/what-is-byte-stuffing-in-computer-networks)](https://www.tutorialspoint.com/what-is-byte-stuffing-in-computer-networks)': are bytes used as a delimiter to mark the end of one frame and the beginning of the next frame.
- '[Stuffed bytes (https://www.tutorialspoint.com/what-is-byte-stuffing-in-computer-networks)](https://www.tutorialspoint.com/what-is-byte-stuffing-in-computer-networks)': also called character-oriented framing, is the escape character (ESC) that is stuffed before every byte in the message with the same pattern as the flag byte. This is done to distinguish actual bytes in a message that is the same pattern as the flag byte.
- In Radarcape Beast format data stream, '0x1a' is used as the flag byte and stuffed byte.

In [ ]:

```
            # process the buffer until the last divider <esc> 0x1a and reset buffer wi
            while i < len(buffer):

                # Check if we are at a flag byte:
                if (buffer[i] != 0x1a):
                    # if we are not, append
                    completed_msg.append(buffer[i])
                    # try:
                        # completed_msg.append(buffer[i])
                    # except AttributeError as error:
                        # completed_msg = [completed_msg]
                        # completed_msg.extend(buffer[i])

                # we are at a flag byte:
                else:
                    # Check if we have reached the end of the buffer:
                    if (i == len(buffer) - 1):
                        # if so, append
                        completed_msg.append(0x1a)
                    # if we have not yet reached the end and the next byte is also a f
                    # the byte we are at is a 'stuffed' byte:
                    elif (buffer[i+1] == 0x1a):
                        # append, but skip the next byte:
                        completed_msg.append(0x1a)
                        i += 1
                    # if we have not yet reached the end and the next byte is not a fl
                    # we have one complete message:
                    elif len(completed_msg) > 0:
                        completed_msg_list.append(completed_msg)
                        completed_msg = []
                        # Update counter:
                        main_message_count = main_message_count + 1

                i += 1
```

Logic Flow to group bytes to form complete messages:

1. We set a loop to process the whole length of the buffer
2. We first check if we are at a flag byte.

    - If we are not, we can just append the byte safely because it is a 'normal' byte in a message.
3. If we are at a flag byte, we have to first check if we have processed till the end of the buffer.
    - If we have processed until the end of the buffer, we treat the byte as a 'flag byte'.
      Note that this will cause a loss of two messages at maximum, if it so happens that a 'stuffed byte' is exactly at the end of the buffer. Although the probability of this happening is very low, there does not seem to be a way around it if 'real-time' analysis is desired.
4. If we are at a flag byte and we have not processed till the end of the buffer, we then check if the next byte is also a 'flag byte'. (Note: The logic flow of the code is such that doing this does not occur any 'index out-of-range' errors).
   If it is also a 'flag byte', it means that we are currently at a 'stuffed byte'. We then append the current byte (which would be '1a') but skip the next byte.
5. Lastly, if we have not yet reach the end of the buffer and the next byte is not a flag byte, it means that the bytes that we are "holding onto" (in complete_msg) is a complete message. We then append the 'flag byte', clear the bytes that we are "holding onto" (in complete_msg), and add the complete message into our list of complete messages (in completed_msg_list).
   We also update the counter by 1, symbolising that we have now grouped 1 more complete message.

In [ ]:

```python
            # save the remainder for next reading cycle, if not empty:
            if len(completed_msg) > 0:
                remainder = []
                for i, m in enumerate(completed_msg):
                    if (m == 0x1a) and (i < len(completed_msg)-1):
                        # rewind 0x1a, except when it is at the last bit
                        remainder.extend([m, m])
                    else:
                        remainder.append(m)
                buffer = [0x1a] + remainder
            else:
                # Else empty, reset buffer
                buffer = []

            # Messages segmented - Pipe list of completed messages (in list form):
            try:
                parent_end.send(completed_msg_list)
            except Exception as e:
                errorHandler('1d')

        except KeyboardInterrupt:
            run_parent = False
            print("***************")
            # Print minimum buffer margin upon exit i.e. ideally we always want some m
            time_now = str(datetime.utcnow())
            print("%s | Minimum buffer margin at socket: %d" %(time_now, min_margin))
            print("%s | Main message count: %d" %(time_now, main_message_count))
            print("%s | Main process terminated" %(time_now))
            print("***************")
            # Returns control to the top of the loop:
            continue

        except Exception as e:
            try:
                sock = connect()
            except Exception as e:
                errorHandler('1e')
```

The previous code snippet groups the messages that have all their bytes in a single buffer. However, we also have to take into account the 'bytes of the previous message' that may also appear at the start, and/ or 'bytes of the next message' that may also appear at the end. This occurs because we are dealing with a byte steam, and bytes that come in do not only come in 'complete groups'.

- The above code snippet handles this issue by 'saving the remainders' in a list, which would then be used in conjunction with the next incoming buffer to group messages.
- Although I am not entirely sure of how it works, the code snippet was extracted from pyModeS.
  However, it was important for the function to still perform as expected. Refer to Appendix A to see a check done on its validity with sample buffers (from collected data).

At a higher level up to this point, an incoming buffer of bytes is grouped whenever possible into complete messages, with the remainder stored for the next buffer for grouping again.
The output from this main function is a list of complete messages (in list form), which is then piped to the next process in the pipeline for data extraction and decoding.

Decoder process

The decoder process receives list of complete messages (in list form) from parent process, extract and decodes for desired information, and pipes it to the Endpoint process.

In [ ]:

```python
"""Process that extracts and decodes complete messages"""
def decoder(decode_start, decode_end):
    '''
    <esc> "1" : 6 byte MLAT timestamp, 1 byte signal level, 2 byte Mode-AC
    <esc> "2" : 6 byte MLAT timestamp, 1 byte signal level, 7 byte Mode-S short frame
    <esc> "3" : 6 byte MLAT timestamp, 1 byte signal level, 14 byte Mode-S long frame
    <esc> "4" : 6 byte MLAT timestamp, status data, DIP switch configuration settings
    <esc><esc>: true 0x1a
    <esc> is 0x1a, and "1", "2" and "3" are 0x31, 0x32 and 0x33
    timestamp: wiki.modesbeast.com/Radarcape:Firmware_Versions#The_GPS_timestamp
    '''

    # Initialize variables:
    run_decoder = True          # process control flag
    decoder_message_count = 0   # keep count number of decoded messages

    while run_decoder:
        try:

            # Receive from parent process:
            try:
                completed_msg_list = decode_start.recv()
            except Exception as e:
                errorHandler('2a')

            messages_to_write = []  # List to contain messages (msg_to_write, in list 
            msg_to_write = []       # Individual messages in list form

            # Loop through list of messages (in list form):
            for message in completed_msg_list:

                # Decode message type and extract payload from message:
                msgtype = message[0]
                if msgtype == 0x32:
                    # Mode-S Short Message, 7 byte, 14-len hexstr
                    payload = ''.join('%02X' % i for i in message[8:15])
                elif msgtype == 0x33:
                    # Mode-S Long Message, 14 byte, 28-len hexstr
                    payload = ''.join('%02X' % i for i in message[8:22])
                    # Added to handle Mode-AC:
                elif msgtype == 0x31:
                    # Mode-AC, 9 byte, 18-len hexstr
                    payload = ''.join('%02X' % i for i in message[8:10])
                else:
                    # Other message tupe
                    continue

                # incomplete message, control returned to top of for loop:
                if len(payload) not in [4, 14, 28]:
                    continue
                # # more secured alternative (only for mode S):
                # df = df(payload)
                # # skip incomplete message
                # if df in [0, 4, 5, 11] and len(payload) != 14:
                    # continue
                # if df in [16, 17, 18, 19, 20, 21, 24] and len(payload) != 28:
                    # continue
```

This process does the following, with the loop set to run continuously:

1. Receive a list of complete messages (in list form) from parent
2. For each individual complete messages (in list form), extract and inspect the bytes to determine the message type (mode AC or mode S)
3. Depending on the message type, extract the bytes that represents the 'payload'. Information on the number and position of these bytes to extract follows the one in pyModeS, which is derived from the original documentation from the creators of Radarcape (https://wiki.jetvision.de/wiki/Mode-S_Beast:Data_Output_Formats).
   Note: This 'payload', which originates from the aircraft, does not carry timestamp information. This is supplied by Radarcape, which is later extracted seperately.
4. At the end of this code snippet, 'payload' is a hexadecimal string
5. Significance of '%02x' (https://stackoverflow.com/questions/14678132/python-hexadecimal)

In [ ]:

```
                    # Extract hexadecimal string signifying the timestamp from full_message
                    # Method is common for all three message types above
                    time_hex = ''.join('%02X' % i for i in message[1:7])

                    # Call function to decode for timestamp in decimal/ float form:
                    ts = num_timestamp(time_hex)

                    # Extract full message:
                    full_message = ''.join('%02X' % i for i in message[:])

                    # Call function to decode for downlink format:
                    downlink = df(payload)  # Garbage for mode A-C

                    # Get Internet timestamp:
                    utc_now = datetime.utcnow()
                    utc_midnight = utc_now
                    localtime = (utc_now - utc_midnight.replace(hour=0, minute=0, second=0
```

Update: This code for timestamp decoding is erroneous - refer to Appendix B for a possible soluion to decode timestamps.

The above code snippet extracts and decodes for the desired information. This is a custom and original code snippet; and information on the number and position of these bytes to extract was derived from the original documentation by the creators of Radarcape (https://wiki.jetvision.de/wiki/Mode-S_Beast:Data_Output_Formats).

1. Timestamp:
   - Join the bytes 1 through 7 (6 bytes) which represents the timestamp into a hexadecimal string. Note that byte 0 represents the message type (used above) and has no use here.
   - Call our custom function and send the hexadecimal string for it to decode and return a decimal timestamp, in the form of 'seconds . nanoseconds' from UTC midnight of the day.
2. Join all the bytes in the message (full_message) for debugging purposes.
3. Call our function (sourced from pyModeS) and send the hexadecimal string for it to decode and return the downlink format as a string. Note that if the message sent is in mode AC format, the string returned is garbage.

Notes on Hexadecimal timestamp from Radarcape and Internet UTC time (https://en.wikipedia.org/wiki/Network_Time_Protocol):

1. The Hexadecimal timestamp as supplied by the Radarcape is obtained via GPS. In addition to using a GPS dongle, the user also has the set the appropriate mode in Radarcape. [The GPS system (https://timetoolsltd.com/gps/what-is-the-gps-clock/)](https://timetoolsltd.com/gps/what-is-the-gps-clock/) is a constellation of 24 orbiting satellites. Each GPS satellite has an integral atomic clock which is synchronized periodically to a ground based master clock maintained by the U.S. Naval Observatory (USNO). USNO maintains synchronization of the entire GPS system to international standards. The system is primarily intended to provide a precise positioning and navigation service. However, very accurate time information is continuously transmitted by the satellites. By receiving these transmissions of 'highly accurate time' using a GPS dongle, the Radarcape gets a copy of and outputs this as timestamp.
2. [Synchronizing a client to a network server (http://www.ntp.org/ntpfaq/NTP-s-algo.htm)](http://www.ntp.org/ntpfaq/NTP-s-algo.htm) consists of several packet exchanges where each exchange is a pair of request and reply. When sending out a request, the client stores its own time (originate timestamp) into the packet being sent. When a server receives such a packet, it will in turn store its own time (receive timestamp) into the packet, and the packet will be returned after putting a transmit timestamp into the packet. When receiving the reply, the receiver will once more log its own receipt time to estimate the travelling time of the packet. The travelling time (delay) is estimated to be half of "the total delay minus remote processing time", assuming symmetrical delays.

Therefore, as a sanity check; we also obtained the internet UTC time in the form of 'seconds . microseconds' from UTC midnight of the day. This is to validate our hyphothesis that assuming the decoded timestamp (as returned as a decimal timestamp in exchange to us sending it a hexadecimal string) and internet UTC time (in the form of 'seconds . microseconds' from UTC midnight of the day) to be correct, the latter should be larger than the former and within the range of 0 to 1 seconds. This is because it takes some time for our script to process our messages and decode for the timestamps. However in my testings, I found the decoded timestamp to be very consistently 0.5 seconds larger than the internet UTC time. This is illogical because it would appear that the GPS timestamps are 'of the future'. More troubleshooting and likely debugging is necessary here.

Update: This code for timestamp decoding is erroneous - refer to Appendix B for a possible soluion to decode timestamps.

Let's now take a look at our first custom function, num_timestamp:

In [11]:

```python
"""Function to convert time_hex to a readable decimal timestamp"""
def num_timestamp(time_hex):
    scale = 16                    # base 16, hexadecimal
    num_of_bits = 48              # 6 bytes = 6 x 8 = 48 bits

    # Convert to binary with leading zeros:
    binary_data = bin(int(time_hex, scale))[2:].zfill(num_of_bits)

    # Extract and convert for seconds and nanoseconds:
    s = binary_data[:18]
    ns = binary_data[18:]
    s = int(s, 2);
    ns = int(ns, 2);
    ts = str("%d.%d" %(s,ns))   # string concatonation

    # Typecast string to float/ decimal:
    ts_float = float(ts)
    return ts_float
```

1. This custom function first converts the hexastring into an integer, taking hexadecimal's base 16 into account.

2. This integer is then converted into binary form, with '0's appended on the left up to 48 bits.
3. The bits used for 'seconds' and 'nanoseconds' are then extracted and converted into integers of base 2.
4. These integers are then concatonated into the form of seconds + . + nanoseconds and finally typecasted to a string.

The end result returned from this custom function is a decimal float (base 10) of the GPS timestamp.

We then take a look at our second custom function, 'downlink'. For this function to work, it requires 2 other sub-functions; hex2bin and bin2int, effectively becoming a set of necessary functions to obtain the downlink format. This is what was used by pyModeS, and is assumed to be valid.

In [ ]:

```python
"""Set of necessary functions to obtain downlink format"""
def hex2bin(hexstr):
    """Convert a hexdecimal string to binary string, with zero fillings. """
    num_of_bits = len(hexstr) * 4
    binstr = bin(int(hexstr, 16))[2:].zfill(int(num_of_bits))
    return binstr
def bin2int(binstr):
    """Convert a binary string to integer. """
    return int(binstr, 2)
def df(msg):
    """Decode Downlink Format value, bits 1 to 5."""
    msgbin = hex2bin(msg)
    return min( bin2int(msgbin[0:5]) , 24 )
```

We refer back and continue with our decoder process:

In [ ]:

```
                # First, check if len = 4 (mode-AC):
                if (len(payload)==4):
                    # Format list to contain data:
                    # squawk for mode AC taken to be just 'payload', a 2 byte message
                    msg_to_write = [full_message, time_hex, ts, localtime, downlink, p
                    # Append to list of formatted messages for writing:
                    messages_to_write.append(msg_to_write)

                # Call function to decode for squawk if DF5, 17, 21. For more supporte
                elif (downlink in [5, 17, 21]):
                    squawk = idcode(payload)
                    msg_to_write = [full_message, time_hex, ts, localtime, downlink, s
                    # Append to list of formatted messages for writing:
                    messages_to_write.append(msg_to_write)

                # Not interested - no squawk code:
                else:
                    pass

                # Update counter:
                decoder_message_count = decoder_message_count + 1

            # Messages decoded - Pipe list of decoded messages (in list form):
            try:
                decode_end.send(messages_to_write)
            except Exception as e:
                errorHandler('2b')

        # process termination control:
        except KeyboardInterrupt:
            run_decoder = False
            print("***************")
            time_now = str(datetime.utcnow())
            print("%s | Decoder message count: %d" %(time_now, decoder_message_count))
            print("%s | Decoder process terminated" %(time_now))
            continue    # Returns control to the top of the loop
```

The above code snippet deals with constructing our list with our extracted desired information from a single message (msg_to_write).
These messages is then appended to another list, resulting in a list of lists (messages_to_write).
Logic flow:

1. We first check if our message is in mode AC. Here, we do this by looking at the length of the message (mode AC messages are of length 4 hexa = 2 bytes). If our message is indeed mode AC, we do not attempt to call idcode() to decode for a squawk code, since it holds no meaning. Instead, we assume the payload to be containing the squawk code (although in reality this can be either the altitude for mode A or sqawk code for mode C). Since we do not know if the 'interrogator' asked for the squawk code or altitude from the aircraft, we are unable to differentiate the two.
2. Because of the 'fliter' for the length of messages done above (in a few code snippets above to ensure that messages sent are complete), any messages left after step 1 is either of length 14 (for mode S short) or length 28 (for mode S long).
3. *According to Oliver, only mode S messages that are of Downlink 5, 17 or 21 contain the squawk code.* Hence, we set up another 'filter' in the form of list and compare our squawk code. If the squawk code for a particular message is either 5, 17 or 21, we pass the 'payload' to our idcode() function to extract and decode for the squawk code.

4. If our message is in either of the following forms, we then construct our list (msg_to_write) in the following way:
    - Mode AC: **[**full_message (hexa decimal string including Radarcape Beast timestamp)**,** time_hex (portion of hexadecimal string sent to function num_timestamp to decode for float decimal timestamp)**,** ts (float decimal timestamp received back from function num_timestamp)**,** localtime (float decimal time, internet UTC time)**,** downlink (string, downlink format)**,** msg (hexadecimal string of payload)**]**
    - Mode S **and** of either Downlink format 5, 17, 21: **[**full_message (hexa decimal string including Radarcape Beast timestamp)**,** time_hex (portion of hexadecimal string sent to function num_timestamp to decode for float decimal timestamp)**,** ts (float decimal timestamp returned from function num_timestamp)**,** localtime (float decimal time, internet UTC time)**,** downlink (string, downlink format)**,** Squawk code (hexadecimal string of squawk code returned from function idcode)**]**

If the message is either one of the above, we 'accept' the message into our list of messages (messages_to_write, in the form of a list). Else, the payload does not contain the squawk code and hence we ignore it.
For either case, the decoder is considered to have decoded a message and hence we update our message counter.
After the decoder finishes processing the list of messages (in list form) that it received from the parent process, it then pipes messages_to_write to the Endpoint process.

Before we move onto the endpoint process, the code snippet below shows the idcode() function. My understanding on the function is still unclear. It was extracted from pyModeS and is assumed to be valid.

In [ ]:

```python
"""Function to obtain Squawk code from message"""
def idcode(msg):
    mbin = hex2bin(msg)
    C1 = mbin[19]
    A1 = mbin[20]
    C2 = mbin[21]
    A2 = mbin[22]
    C4 = mbin[23]
    A4 = mbin[24]
    # _ = mbin[25]
    B1 = mbin[26]
    D1 = mbin[27]
    B2 = mbin[28]
    D2 = mbin[29]
    B4 = mbin[30]
    D4 = mbin[31]
    byte1 = int(A4+A2+A1, 2)
    byte2 = int(B4+B2+B1, 2)
    byte3 = int(C4+C2+C1, 2)
    byte4 = int(D4+D2+D1, 2)
    return str(byte1) + str(byte2) + str(byte3) + str(byte4)
```

Endpoint process

In [ ]:

```python
"""Process that update filenames and write messages to file"""
def endpoint(endpoint_start):
    prev_fn = "dummy.txt"   # Dummy filename for initialisation
    run_endpoint = True
    endpoint_message_count = 0

    while run_endpoint:
        try:

            messages_to_write = []

            try:
                messages_to_write = endpoint_start.recv()   # receiving list of messag
            except Exception as e:
                errorHandler('3a')

            for message in messages_to_write:
```

For starters, the process initialises a few variables and starts the loop and runs it continuously. In this loop, the processer tries to receive the list of messages (in list form) piped from the decoder process. It then loops through the list and deals with each message.

In [ ]:

```python
                try:

                    # Routine to update filenames (using seconds):
                    # Extract timestamp and set it as temporary filename:
                    time_for_fn = message[2]    # float
                    # rounds it down using int (floor) and converts it to str:
                    filename = str(int(time_for_fn))
                    filename = filename+".txt"
```

It was expressed to be a desire for messages that are received in the same second to be saved together onto the same text file. To do this:

1. We first extract the decoded timestamp from each message via its index i.e. message[2] since we have previously defined in the decoder function that this index in the list would store the decoded timestamp, in float decimal form.
2. With this decoded timestamp in float decimal, we typecast it to an integer, which rounds it down to the nearest integer; effectively 'ignoring' the decimal places.
    - e.g. float(1999.999999999) becomes int(1999)
3. We then typecast the resulting integer into a string in order to concatonate ".txt" to it to create a proper usable textfile name to save messages in.

In [ ]:

```
                    # if one second has passed, update filename:
                    if (filename!=prev_fn):
                        try:
                            prev_fn.close() # close the previous file
                        except:             # Expected error on first loop
                            pass
                        prev_fn = filename  # Update the old filename
                        file = open(prev_fn,"a")
```

The code snippet above keeps filenames updated, with an emphasis on efficiency:

1. First, it checks if the filename of the file that it is currently writing to needs to be updated.
2. If the filename it is supposed to save messages in needs to be updated (according to the decoded timestamp), it will attempt to close the file that it has been writing to.
   - An error to occur in the first 'loop/ instance' of this process is expected. This is because the .close() statement would be addressed to a file that has not been created/ does not exist. Hence, to bypass this error and allow the script to continue executing, we placed it in a try/ except clause.
3. After closing the file that it has been writing to, it will then update the filename, and finally open a file with the new filename to write messages to.

In [ ]:

```
                    # Was a list, cast it into a string:
                    line = str(message)
                    # Remove '[' and ']' due to typecasting from list:
                    line_to_write = line[1::][:-1:]
                    # write to file with updated filename:
                    file.write("%s\n" %line_to_write)
                    # Update counter:
                    endpoint_message_count = endpoint_message_count + 1

                # Error - try to extract data for debugging:
                except Exception as e:
                    errorHandler('3b')

        except KeyboardInterrupt:
            run_endpoint = False
            print("***************")
            time_now = str(datetime.utcnow())
            print("%s | Endpoint message count: %d" %(time_now, endpoint_message_count
            print("%s | Endpoint process terminated" %(time_now))
            continue    # Returns control to the top of the loop
```

The code snippet above does final pre-write formatting and writes lines to the textfile with an updated filename.

1. To start, each of the messages (in list form) in the message_to_write list is casted as a string.
2. Then, we remove the leading '[' and trailing ']' from the string - they are unnecessary but present here because it used to be a list.
3. We then write the line to the textfile with an updated filename and to include a newline character at the end for textfile readibility.
4. As expected, for each message that was written into the textfile, the counter at the Endpoint updates to keep count of the messages that it was written.

Miscellaneous: ErrorHandler function

An important aspect of the project is being accountable for every message and to as much as possible prevent losing data/ dropping messages.

- In order to facilitate this, a specialised function was created to handle these errors i.e. writing these errors to file. * To use this function, simply call this function from the main body of code with the corresponding error code. The error message corresponding to the error code is then printed onto the textfile.
- This function also acquires and prints the internet UTC time as feedback to users the time at which these errors occurred for debugging purposes.

This function was implemented to keep the main body of the code neater and more readable. It is also expected to help with scability in the future, since accounting for errors is now easier and adding new errors to report is as simple as adding it into this function, and calling this function with the corresponding error code.

In [ ]:

```python
"""Handles errors and print them to file"""
def errorHandler(code):
    """
    args: code in string format
    """
    error_timestamp = str(datetime.utcnow())

    if (code == '1a'):
        error_msg = "ERROR : Socket connection error. Reconnecting..."
    elif (code == '1b'):
        error_msg = "ERROR : Message length received is zero at socket"
    elif (code == '1c'):
        error_msg = "ERROR : There is no margin at 16384 buffer length"
    elif (code == '1d'):
        error_msg = "ERROR : Parent send error"
    elif (code == '1e'):
        error_msg = "ERROR : Parent receive error"
    elif (code == '2a'):
        error_msg = "ERROR : Decode receive error"
    elif (code == '2b'):
        error_msg = "ERROR : Decode send error"
    elif (code == '3a'):
        error_msg = "ERROR : Endpoint receive error"
    elif (code == '3b'):
        error_msg = "ERROR : Endpoint write error"

    error_msg_printed = ("%s : %s\n" %(error_timestamp, error_msg))

    with open("0_Log_Errors.txt", "a") as text_file:
        text_file.write(error_msg_printed)

    return
```

Appendix A:

Validity-check for function to group bytes into complete message (in list form) and to keep remainder for next cycle:

In [ ]:

```python
def num_timestamp(time_hex):
    """
    Function to convert between time_hex (a portion of data) to a readable timestamp
    args:
        time_hex
    outputs:
        padded timestamp in string format
    """
    # Function that acquires the timestamp from data
    scale = 16                  # equals to hexadecimal
    num_of_bits = 48            # 48 bits in our case 6 X 8?
    binary_data = bin(int(time_hex, scale))[2:].zfill(num_of_bits)
    s = binary_data[:18]
    ns = binary_data[18:]
    s = int(s, 2);
    ns = int(ns, 2);
    ts = str("%d.%d" %(s,ns))   # Data timestamp set here
    ts = '{:<20}'.format(ts)    # Pad to 20 spaces for alignment to check DF.
    return ts

"""minic data stream buffer:"""

"""logical tests"""
# complete messages at a time:
# buffer1 = b'\x1a1\x19\xfc\x11\xf4\xd6\x96g\x01\x10\x1a1'              # complete sin
# buffer1 = b'\x1a1\x19\xfc\x11\xf4\xd6\x96g\x01\x10\x1a1\x19\xfc\x11'  # complete sin
# buffer1 = b'\x19\xfc\x11\x1a1\x19\xfc\x11\xf4\xd6\x96g\x01\x10\x1a1'  # complete sin
# buffer2 = b'\x1a1\x19\xfc\x12\x1a\x1a\xd9\x1eAQ$\x1a1'                # complete sin
# buffer2 = b'\x1a1\x19\xfc\x12\x1a\x1a\xd9\x1eAQ$\x1a1\x19\xfc\x11'    # complete sin
# buffer2 = b'\x19\xfc\x11\x1a1\x19\xfc\x12\x1a\x1a\xd9\x1eAQ$\x1a1'    # complete sin
"""test sets, together validates functionality"""
# test set 1:
buffer1 = b'\x1a1\x19\xfc\x11\xf4'                             # no messages
buffer2 = b'\xd6\x96g\x01\x10\x1a1\x19\xfc\x12\x1a\x1a\xd9'    # 1 + messages
buffer3 = b'\x1eAQ$\x1a1\x19\xfc\x12.\xa7\xfaX'                # 2 + messages
# test set 2:
# buffer1 = b'\x1a1\x19\xfc\x11\xf4\xd6\x96g\x01\x10\x1a1\x19\xfc\x12\x1a\x1a\xd9\x1eA

buffers_all = [buffer1, buffer2, buffer3]



buffer = []

for buffer_raw in buffers_all:
    # Referenced - initialise and empty list after sending at the end of each loop:
    completed_msg_list = []
    completed_msg = []
    i = 0
    main_message_count = 0
    buffer.extend(buffer_raw)

    # process the buffer until the last divider <esc> 0x1a and reset buffer with remai
    while i < len(buffer):
        if (buffer[i:i+2] == [0x1a, 0x1a]):
            completed_msg.append(0x1a)
            i += 1
        elif (i == len(buffer) - 1) and (buffer[i] == 0x1a):
            # special case where the last bit is 0x1a
```

```python
                completed_msg.append(0x1a)
            elif buffer[i] == 0x1a:
                if i == len(buffer) - 1:
                    # special case where the last bit is 0x1a
                    completed_msg.append(0x1a)
                elif len(completed_msg) > 0:
                    completed_msg_list.append(completed_msg)
                    completed_msg = []
            else:
                completed_msg.append(buffer[i])
            i += 1

    # save the remainder for next reading cycle, if not empty:
    if len(completed_msg) > 0:
        remainder = []
        for i, m in enumerate(completed_msg):
            if (m == 0x1a) and (i < len(completed_msg)-1):
                # rewind 0x1a, except when it is at the last bit
                remainder.extend([m, m])
            else:
                remainder.append(m)
            # remainder.append(m)
        buffer = [0x1a] + remainder  # error here previously
    else:
        # Else empty, reset buffer
        buffer = []
    print("Completed Message List:")
    print(completed_msg_list)
    print("Completed message")
    print(completed_msg)
    print("Reminder:")
    print(remainder)

    """GET TIMESTAMP"""
    # Decode messages in list and format them for writing to file:
    messages_to_write = []  # List of msg_to_write
    msg_to_write = ''       # Temp string variable of formatted messages
    for mm in completed_msg_list:

        # Decode message type and extract msg from data:
        msgtype = mm[0]
        if msgtype == 0x32:
            # Mode-S Short Message, 7 byte, 14-len hexstr
            msg = ''.join('%02X' % i for i in mm[8:15])
        elif msgtype == 0x33:
            # Mode-S Long Message, 14 byte, 28-len hexstr
            msg = ''.join('%02X' % i for i in mm[8:22])
            # Added to handle Mode-AC:
        elif msgtype == 0x31:
            # Mode-AC, 9 byte, 18-len hexstr
            msg = ''.join('%02X' % i for i in mm[8:10])
        else:
            # Other message tupe
            continue

        # # incomplete message, control returned to top of for loop:
        # if len(msg) not in [4, 14, 28]:
        #     continue

        # Extract timestamp from data (method common for all three message types above
        time_hex = ''.join('%02X' % i for i in mm[1:7])

        # Decode extracted timestamp:
        ts = num_timestamp(time_hex)

        print(ts)
```

The above validity test code was extracted from the original main code and was set up to run in isolation.

1. To validate that the code groups bytes into complete messages (in list form) and that it keeps the remainders to be used in the next cycle correctly; buffer test sets (from actual collected data) were synthetically generated and fed into the function using a loop.
2. These buffer test sets were initialised at the start, and have been succesful in their tests.
3. The buffer test set that best represents actual operation (left uncommented in the code above) was chosen for an illustration/ demo here. The details of the buffers in this buffer test set is explained below:
    - buffer1: contains only parts of the first message.
    - buffer2: when combined with buffer1 by appending buffer1 in front, it contains the first full message and parts of the second message.
    - buffer3: when combined with both buffer1 and buffer2 by appending them in front, it contains the first and second full messages and parts of a third message.

In [ ]:

```
PS C:\Users\user\Desktop> python ts_check_final.py
Completed Message List:
[]
Reminder:
[49, 25, 252, 17, 244]
Completed Message List:
[[49, 25, 252, 17, 244, 214, 150, 103, 1, 16]]
Reminder:
[49, 25, 252, 18, 26, 26, 217]
26608.301258390
Completed Message List:
[[49, 25, 252, 18, 26, 217, 30, 65, 81, 36]]
Reminder:
[49, 25, 252, 18, 46, 167, 250, 88]
26608.303749406
```

The result is the expected complete messages and the expected remainders after every cycle.

Profiling:

Why the need for profiling:

1. Due to the 'pipeline' architecture of the code; the time at which a process is able to start is actually dependent on the process before it. i.e.
    - the decoder process can only start after the parent process is done with grouping a single buffer and pipes it over.
    - the writer process can only start after the decoder process is done with decoding a single buffer and pipes it over.
2. This would mean that the entire script is subjected to a 'only-increasing' time delay and that the script's performance is dependent on the 'slowest' proess (like the weakest link in a chain).

3. Since it was important for the overall rate at which messages enter and exit the entire pipeline to be able to keep up with the incoming message rate (as established previously), it was important to find out the bottleneck(s) in the pipeline and to attach a repeatly-measurable and meaningful metric to its speed.
    - This needs to be done before actual deployment, since measurements upon deployment would be too late (data overflow and lost).
    - This will also share some insights on the process that attention should be spent on in future improvementments for maximum efficiency. e.g. if the writing process of files is already faster than the decoder process, it would be less meaningful to spend time on improving the writing process than it is on the decoder process; since the performance of the pipeline is dependent on the slowest process.

In [ ]:

```python
"""
simulation to track maximum speeds, with architecture based on ver2.
"""

# Importing required libraries:
# tcp client:
import os
import sys
import socket
import time
import datetime
from threading import Thread
# Multiprocessing:
import multiprocessing

def hex2bin(hexstr):
    """Convert a hexdecimal string to binary string, with zero fillings. """
    num_of_bits = len(hexstr) * 4
    binstr = bin(int(hexstr, 16))[2:].zfill(int(num_of_bits))
    return binstr

def bin2int(binstr):
    """Convert a binary string to integer. """
    return int(binstr, 2)

def df(msg):
    """Decode Downlink Format value, bits 1 to 5."""
    msgbin = hex2bin(msg)
    return min( bin2int(msgbin[0:5]) , 24 )

def idcode(msg):
    """
    Args:
        msg (String): 28 bytes hexadecimal message string
    Returns:
        string: squawk code
    """
    if df(msg) not in [5, 21, 17]:
        raise RuntimeError("Message must be Downlink Format 5, 21 or 17.")
    mbin = hex2bin(msg)
    C1 = mbin[19]
    A1 = mbin[20]
    C2 = mbin[21]
    A2 = mbin[22]
    C4 = mbin[23]
    A4 = mbin[24]
    # _ = mbin[25]
    B1 = mbin[26]
    D1 = mbin[27]
    B2 = mbin[28]
    D2 = mbin[29]
    B4 = mbin[30]
    D4 = mbin[31]
    byte1 = int(A4+A2+A1, 2)
    byte2 = int(B4+B2+B1, 2)
    byte3 = int(C4+C2+C1, 2)
    byte4 = int(D4+D2+D1, 2)
    return str(byte1) + str(byte2) + str(byte3) + str(byte4)

```

```python
def num_timestamp(time_hex):
    """
    Function to convert between time_hex (a portion of data) to a readable timestamp
    args:
        time_hex
    outputs:
        padded timestamp in string format
    """
    # Function that acquires the timestamp from data
    scale = 16                   # equals to hexadecimal
    num_of_bits = 48             # 48 bits in our case 6 X 8?
    binary_data = bin(int(time_hex, scale))[2:].zfill(num_of_bits)
    s = binary_data[:18]
    ns = binary_data[18:]
    s = int(s, 2);
    ns = int(ns, 2);
    ts = str("%d.%d" %(s,ns))    # Data timestamp set here
    ts = '{:<20}'.format(ts)     # Pad to 20 spaces for alignment to check DF.
    return ts

def decoder(decode_start, decode_end):
    # initialise number of decoded messages:
    decoder_message_count = 0

    # List of downlink formats we are interested in:
    DF_filter = ['28','29','2A','2B','2C','2D','2E','2F','A8','A9','AA','AB','AC','AD'

    # Receive from parent process:
    try:
        completed_msg_list = decode_start.recv()
        time_two = time.time()
        print("simulated message list attempted and successfully received by decoder a
        f = open("simulation_log.txt","a")
        f.write("simulated message list attempted and successfully received by decoder
        f.close()
    except Exception as e:
        print("error occurred when attempting to receive message list by decoder, %s,

    # Decode messages in list and format them for writing to file:
    messages_to_write = []  # List of msg_to_write
    msg_to_write = ''       # Temp string variable of formatted messages
    for mm in completed_msg_list:

        # Decode message type and extract msg from data:
        msgtype = mm[0]
        if msgtype == 0x32:
            # Mode-S Short Message, 7 byte, 14-len hexstr
            msg = ''.join('%02X' % i for i in mm[8:15])
        elif msgtype == 0x33:
            # Mode-S Long Message, 14 byte, 28-len hexstr
            msg = ''.join('%02X' % i for i in mm[8:22])
            # Added to handle Mode-AC:
        elif msgtype == 0x31:
            # Mode-AC, 9 byte, 18-len hexstr
            msg = ''.join('%02X' % i for i in mm[8:10])
        else:
            # Other message tupe
            continue

        # incomplete message, control returned to top of for loop:
        if len(msg) not in [4, 14, 28]:
            continue

        # Extract timestamp from data (method common for all three message types above
        time_hex = ''.join('%02X' % i for i in mm[1:7])

        # Decode extracted timestamp:
        ts = num_timestamp(time_hex)

        # Format messages for writing:
        # Decode for message downlink:
        data = ("%s|%s" % (ts, msg))
        # downlink = df(data[21:])                    # Garbage for mode A-C
        downlink = df(msg)
        dl_double = '{:>2}'.format(downlink)        # Pad for readibility

        # Check if it is len = 4 first (mode-AC):
        if (len(msg)==4):
            msg_to_write = ("%s|%s|%s|%s" %(ts, msg, dl_double, msg))
            # Append to list of formatted messages for writing:
            messages_to_write.append(msg_to_write)

        # Decode for squawk if DF5, 17, 21. For more supported functions, see 'extra':
        elif (data[21:23] in DF_filter):
            squawk = idcode(data[21:])
            msg_to_write = ("%s|%s|%s|%s\n" %(ts, squawk, dl_double, msg))
            # Append to list of formatted messages for writing:
            messages_to_write.append(msg_to_write)

        # Not interested - no squawk code:
        else:
            pass

        # Update counter:
        decoder_message_count = decoder_message_count + 1

    time_three = time.time()
    print("finished decoding %d messages at time: %f" %(decoder_message_count, time_th
    f = open("simulation_log.txt","a")
    f.write("finished decoding %d messages at time: %f\n" %(decoder_message_count, tim
    f.close()

    # After loop completion, pipe data:
    time_four = time.time()
    try:
        decode_end.send(messages_to_write)
        print("simulated message list attempted and successfully sent by decoder, star
        f = open("simulation_log.txt","a")
        f.write("simulated message list attempted and successfully sent by decoder, st
        f.close()
    except Exception as e:
        print("error occurred when attempting to send message list by decoder, %s, tim


def endpoint(endpoint_start):
    prev_fn = "dummy.txt"
    endpoint_message_count = 0

    # Initialise/ reset at the end of each loop to store list of messages to write:
    messages_to_write = []

    # Copy list of messages received from decoder to list of messages to write:
```

```
    try:
        messages_to_write = endpoint_start.recv()
        time_five = time.time()
        print("simulated message list attempted and successfully received by writer at
        f = open("simulation_log.txt","a")
        f.write("simulated message list attempted and successfully received by writer
        f.close()
    except Exception as e:
        print("error occurred when attempting to receive message list by writer, %s, t

    # Start writing each message in message list onto file:
    for message in messages_to_write:
        # Routine to update filenames:

        time_for_fn = ''
        filename = ""
        # Extract timestamp and set it as temporary filename:
        time_for_fn = message[:20]
        # Converts timestamp back to floating number, rounds it down using int (floor)
            # and converts to back to str:
        filename = str(int(float(time_for_fn)))
        filename = filename+".txt"

        # if one second has passed, update filename:
        if (filename!=prev_fn):
            try:
                prev_fn.close() # close the previous file
            except:             # Expected error on first loop
                pass
            prev_fn = filename  # Update the old filename
            file = open(prev_fn,"a")

        # write to file with updated filename:
        file.write("%s" %message)

        # Update counter:
        endpoint_message_count = endpoint_message_count + 1

    time_six = time.time()
    print("finished writing %d messages at time: %f" %(endpoint_message_count, time_si
    f = open("simulation_log.txt","a")
    f.write("finished writing %d messages at time: %f\n" %(endpoint_message_count, tim
    f.close()


if __name__ == '__main__':

    # Create pipes:
    decode_start, parent_end = multiprocessing.Pipe(False)
    endpoint_start, decode_end = multiprocessing.Pipe(False)

    # Create new child processes and give them pipe starts and ends:
    decoding_process = multiprocessing.Process(target=decoder, args=(decode_start, dec
    endpoint_process = multiprocessing.Process(target=endpoint, args=(endpoint_start,)

    # Start both processes:
    decoding_process.start()
    endpoint_process.start()

    # import required libraries:
    import ast
```

```python

    # initialise list to hold duplicated messages (in list form)
    simulated_buffer = []

    """configuration, number of times to duplicate message"""
    config_duplicate = 9000000
    duplicate_count = range(config_duplicate)

    """configuration, edit source text file and content to change test message"""
    source = open("config_message.txt", "r", newline="\n")

    # read first line in source file:
    single_message = str(source.readline())  # assert
    single_message = ast.literal_eval(single_message)

    for duplicate in duplicate_count:
        simulated_buffer.append(single_message)

    str_start_info = str(datetime.datetime.utcnow())
    f = open("simulation_log.txt","a")
    f.write("\nnew simulation started at time: %s, with message count of %d\n" %(str_s
    f.close()

    time_one = time.time()
    try:
        parent_end.send(simulated_buffer)
        print("simulated message list attempted and successfully sent by parent, start
        f = open("simulation_log.txt","a")
        f.write("simulated message list attempted and successfully sent by parent, sta
        f.close()
    except Exception as e:
        print("error occurred when attempting to send message list by parent, %s, time
```

In order to do this profiling, the original script was copied and modified. This modified script inherits the architecture of the original script and all its processes but only has a single synthetic buffer as input. This buffer was synthetically generated by choosing a single complete message (in list form) and duplicating it.
This profiling version does not take into account the time required to group bytes from a data stream into complete messages and to keep the 'remainder' for the next cycle. To take this into account, the profiling should start right at the start of the pipeline i.e. with a synthetically generated buffer of bytes to minic a 'TCP socket buffer' (of bytes) - 'pre-grouped' instead of a list of completed messages (in list form) - 'post-grouped'.

In [ ]:

```python
[51, 23, 222, 175, 103, 60, 248, 45, 141, 138, 7, 77, 153, 68, 209, 142, 104, 40, 11,
```

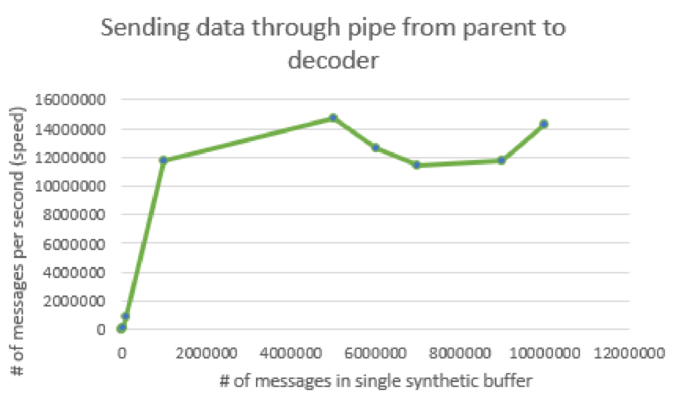
In this profiling version, the above message was chosen for duplication in the buffer.

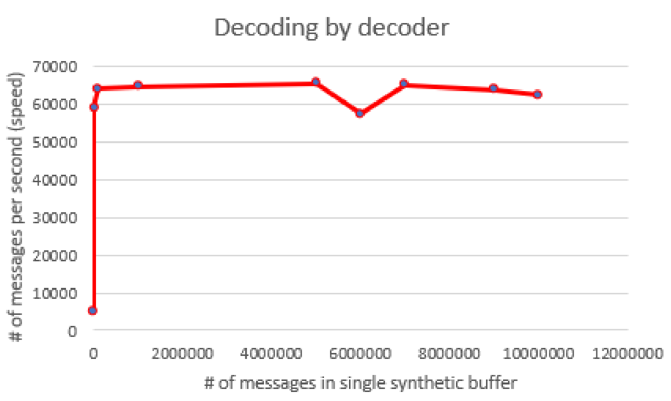
1. It was chosen because it represents a single complete mode S long message, the longest possible type of message.
2. It would be a good indication of the maximum time needed to process a single message, assuming that longer messages require a longer time to process.

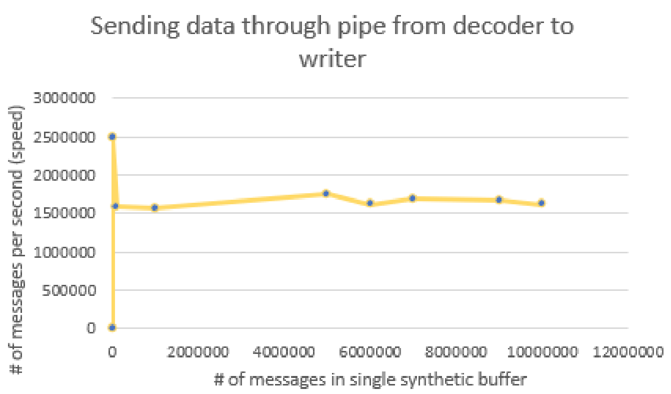
Profiling method:

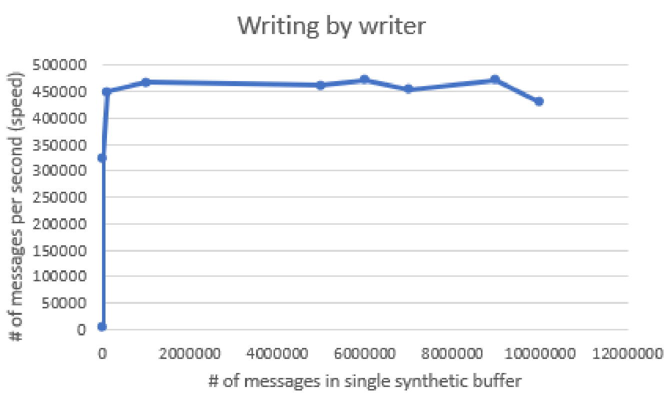
1. The total time spent in each process is logged using the time.time() function.

2. Since we are aware of the number of times the message was duplicated in the buffer, we know the exact number of messages existing in the buffer we sent down the pipeline.
3. Since each message is exactly the same, we can get a good estimate of the time taken to process a single message in each process by dividing (the total time spent in each function) with the (number of messages).
4. From there, we can get a good estimate of the average speed of each process, in terms of the number of messages it is able to process in one second.









Note that speeds quoted here are only theoretical since they can only be reached at very large buffer sizes. In reality, such large buffer sizes can only be reached by waiting for the buffer to fill up - this will necessarily affect how relevant (real-time) the data is, from both the waiting and decoding of the large buffer.

Primary findings:

1. It would appear that the bottleneck is the decoder process - it is the slowest process and processes a maximum of 65,000 messages (long mode S, at a very large buffer size).
2. Notice the flattening of the trendlines - it would appear that maximum speeds have been reached for the decoder, piping of data from decoder to writer and the writer process.

Secondary findings:

1. A safe data limit for single buffers is to contain up to 10 million messages (as tested here). More attempts to tease out greater theoretical speeds by increasing the synthetic buffer size (via duplication) fails. Errors are thrown.
2. Although inheritently the same 'process' working with the same data size, the sending of data through a pipe appears to be faster for 'parent to decoder' than it is for 'decoder to writer'.
   - The only logical explanation is that it is faster to send data through a pipe in 'lists' data structure, since the data structure used in 'parent to decoder' is a list of lists, while the data structure used in 'decoder to writer' is a list of strings **(in version3 and earlier).**

- **In version4 and version4_dev, the data structure used in 'decoder to writer' has been changed to list of lists.**

Changelog - version3 to version4/ version4_dev:

1. Filtering and extractions of partial hexadecimal strings for decoding within functions is now done with list indexing rather than string indexing. This is expected to increase performance.
2. Because of point 1, there is a change in data structure used in 'decoder to writer' pipe from list of strings to list of lists. This is expected to increase performance.
3. Error handling is now done with a specialised function (see Errorhandler function above). It is expected to help with scability in the future, since accounting for errors is now easier and adding new errors to report is as simple as adding it into this function, and calling this function with the corresponding error code.

Appendix B:

Future work

Debug timestamp decoding

Recap:
As a sanity check; we obtained the internet UTC time in the form of 'seconds . microseconds' from UTC midnight of the day. This is to validate our hyphothesis that assuming the decoded timestamp (as returned as a decimal timestamp in exchange to us sending it a hexadecimal string) and internet UTC time (in the form of 'seconds . microseconds' from UTC midnight of the day) to be correct, the latter should be larger than the former and within the range of 0 to 1 seconds. This is because it takes some time for our script to process our messages and decode for the timestamps. However in my testings, I found the decoded timestamp to be very consistently 0.5 seconds larger than the internet UTC time. This is illogical because it would appear that the GPS timestamps are 'of the future'.

Thought process:

1. The current timestamp decoding was deduced from 2 sample test cases (collected data) and matching the results to manual decodings by Oliver.
2. The deduced decoding may not/ is not tested to work reliably for all cases.
   It may even be concluded as erroneous since it does not fulfil the sanity check as mentioned above.
3. Looking rigorouly online for source codes/ explanations on decoding Radarcape Beast timestamps, I landed back at the pyModeS source code. It was then when I noticed that the decoding done in function 'read_skysense_buffer' worked with a 6-byte timestamp, similar to the decoding for the timestamp for Radarcape Beast. This prompted me to investigate the timestamp decoding done here.

In [ ]:

```
"""
            TS field - Time stamp
            Position 15 through 20:
              6 bytes = 48 bits
              Time stamp with fields as:

                Lock Status - Status of internal time keeping mechanism
                Equal to 1 if operating normally
                Bit 47 - 1 bit

                Time of day in UTC seconds, between 0 and 86399
                Bits 46 through 30 - 17 bits

                Nanoseconds into current second, between 0 and 999999999
                Bits 29 through 0 - 30 bits
"""
```

An abstract from pyModeS > extra > tcpclient, read_skysense_buffer

In [ ]:

```
        '''
        <esc> "1" : 6 byte MLAT timestamp, 1 byte signal level,
            2 byte Mode-AC
        <esc> "2" : 6 byte MLAT timestamp, 1 byte signal level,
            7 byte Mode-S short frame
        <esc> "3" : 6 byte MLAT timestamp, 1 byte signal level,
            14 byte Mode-S long frame
        <esc> "4" : 6 byte MLAT timestamp, status data, DIP switch
            configuration settings (not on Mode-S Beast classic)
        <esc><esc>: true 0x1a
        <esc> is 0x1a, and "1", "2" and "3" are 0x31, 0x32 and 0x33

        timestamp:
        wiki.modesbeast.com/Radarcape:Firmware_Versions#The_GPS_timestamp
        '''
```

An abstract from the official [documentation (https://wiki.jetvision.de/wiki/Mode-S_Beast:Data_Output_Formats)](https://wiki.jetvision.de/wiki/Mode-S_Beast:Data_Output_Formats) for Radarcape Beast

In [ ]:

```
                tsbin = self.buffer[i:i+6]
                sec   = ( (tsbin[0] & 0x7f) << 10) | (tsbin[1] << 2 ) | (tsbin[2] >> 6
                nano  = ( (tsbin[2] & 0x3f) << 24) | (tsbin[3] << 16) | (tsbin[4] << 8
                ts = sec + nano*1.0e-9
```

An abstract of the source code at pyModeS > extra > tcpclient, read_skysense_buffer.

4. Upon investigation, this decoding appears to work with the byte strings that represents the timestamp in the 'raw data' - which is also a byte string.
   An example of this raw data is:

   - b'\x19\xfc\x12\x1a\xd9\x1eAQ$', with the prefix 'b' symbolising that it is a string of type byte.

5. A script was written to test this hypothesis.

In [ ]:

```
# binary_data = b'\x19\xfc\x11\xf4\xd6\x96g'
binary_data = b'\x19\xfc\x12\x1a\xd9\x1eAQ$'

# # Extract and convert for seconds and nanoseconds:
# s = binary_data[:18]
# ns = binary_data[18:]
# s = int(s, 2);
# ns = int(ns, 2);
# ts = str("%d.%d" %(s,ns))   # string concatonation

# # Typecast string to float/ decimal:
# ts_float = float(ts)
# return ts_float

sec   = ( (binary_data[0] & 0x7f) << 10) | (binary_data[1] << 2 ) | (binary_data[2] >>
nano  = ( (binary_data[2] & 0x3f) << 24) | (binary_data[3] << 16) | (binary_data[4] <<
ts = sec + nano*1.0e-9

print(ts)
```

Understanding of the workings for this code snippet is still unclear at this point in time.
However, using the same 2 test cases, the outputs printed match those that were manually decoded by Oliver.

Expected outputs:
binary_data (top): 26608.301258390
binary_data (bottom): 26608.303749406

Expected outputs as reference, as manually decoded by Oliver

What we know about this potential solution:

1. The code snippet comes from a seemingly creadible source - a documented and maintained online source code
2. The outputs match those manually decoded by Oliver, an Industrial Expert
3. It may still carry some risk since it was not explicitly mentioned to be for this use case.