Binary File Parser for Mavlink Telemetry Data

Table of Contents

Introduction

This project implements a parser for binary Mavlink telemetry data files. It's designed to handle complex binary structures, interpret various message types, and extract meaningful data, with particular emphasis on GPS coordinates. The parser uses a dynamic approach to understand and decode the binary format, making it adaptable to different versions of Mavlink data structures.

File Structure

The project is organized into several Go files, each handling specific aspects of the parsing process:

- `fileparser.go`: Defines the main `FileParser` interface and factory function.
- `binparser.go`: Implements the `BINParser` struct for high-level binary file parsing.
- `binarydatafilereader.go`: Contains the `BinaryDataFileReader` struct for low-level binary data reading and interpretation.
- `datafileformat.go`: Defines the `DataFileFormat` struct for handling message formats.
- `datafilemessage.go`: Implements the `DataFileMessage` struct for individual parsed messages.
- `gps_interpolated.go:` struct is crucial for handling GPS data and timestamps

Key Components

FileParser Interface

Defines the common interface for all file parsers, allowing for future expansion to other file types.

BINParser

Implements the high-level parsing logic for binary files, focusing on extracting GPS data and creating geometric representations.

BinaryDataFileReader

- Initialization:
    - Memory-maps the binary file for efficient reading.
    - Initializes data structures for message formats, counts, and offsets.
- Main Parsing Loop:

    0. Reads message headers to identify message types.
    1. Uses appropriate unpackers to decode message bodies.
    2. Processes FMT messages to build new unpackers dynamically.
    3. Tracks message counts and offsets for efficient navigation.

- Rewind Functionality:
    0. Allows resetting the reader to the file beginning.
    1. Crucial for multi-pass parsing (e.g., first pass for format discovery, second for data extraction).

DataFileFormat

Represents the structure of each message type within the binary file. It's crucial for creating appropriate unpackers. Holds information like message name, length, format string, and field names. Provides methods to create unpackers based on the format string.

DataFileMessage

Represents individual messages extracted from the binary data, providing methods to access and interpret message contents. Provides methods to access message fields by name. Handles type conversions and applies any necessary scaling factors.

Binary Data Structure

Binary data is a way of storing information using only two states: 0 and 1. Unlike text files where each character is represented by a byte, binary files can use various combinations of bits to represent different types of data efficiently. This makes binary files compact and fast to process, but also harder for humans to read directly.

The binary file is structured as follows:

1. File Header: Contains metadata about the file (if any).
2. Message Sequence: A series of messages, each with:
    - Message Header (typically 3 bytes):
        - Byte 1-2: Message start marker (e.g., 0xA3 0x95)
        - Byte 3: Message type ID
    - Message Body: Variable length, format defined by corresponding FMT message
3. FMT (Format) Messages: Special messages that define the structure of other message types. They include:
    - Message type ID
    - Message length
    - Message name
    - Format string (e.g., "QBIHBcLLefffB")
    - Field names

What is a FMT in binary data files?

In the context of this parser, a FMT (Format) message is a special type of message within the binary file that acts as a data dictionary. It defines the structure and interpretation of other message types in the file. Think of FMT messages as a legend on a map - they tell you how to read and understand the rest of the data.

Structure of a FMT Message

A typical FMT message might contain:

1. Message Type ID (e.g., 128)
2. Message Length (e.g., 89 bytes)
3. Message Name (e.g., "GPS")
4. Format String (e.g., "QBIHBcLLefffB")
5. Field Names (e.g., "TimeUS,Status,GMS,GWk,NSats,HDop,Lat,Lng,Alt,Spd,GCrs,VZ,U")

How FMT is Used in Encoding and Decoding

1. Encoding (Writing the binary file):
    - When creating the binary file, the system first writes FMT messages to define each message type.
    - Subsequent data messages are then written according to these formats.

2. Decoding (Reading the binary file):
    - The parser first reads and interprets all FMT messages.
    - It builds a "data dictionary" (stored in `BinaryDataFileReader.formats`) mapping message types to their formats.
    - When reading data messages, the parser uses this dictionary to know how to interpret the binary data for each message type.

Unpacking Process

- The binary file is memory-mapped for efficient access.
- Initial data structures are set up (e.g., formats, unpackers, offsets).
- The parser scans the file for FMT messages.
- For each FMT message:

    1. It's decoded using a predefined FMT format.
    2. A new `DataFileFormat` object is created and stored.
    3. An `unpacker` function is dynamically created based on the format string.

Building Unpackers

An unpacker is a function that knows how to convert raw binary data into meaningful Go types. For each format character (e.g., 'Q' for uint64, 'f' for float32):

1. The parser determines the corresponding Go type and any necessary scaling.
2. It creates a mini-function to handle that specific type.

These mini-functions are combined to create the full unpacker for each message type.

1. FMT Message Interpretation:
    - The parser first looks for FMT messages in the binary data.
    - Each FMT message is decoded to understand the structure of a specific message type.
2. Unpacker Creation:
    - For each message type defined by an FMT message, an unpacker function is created and stored.
    - The unpacker is built based on the format string in the FMT message.
    - It uses a map (`FormatToUnpackInfo`) to determine how to unpack each byte based on format characters.
3. Dynamic Unpacker Assembly:
    - The `createMessageUnpacker` method in `DataFileFormat` dynamically creates an unpacker function.

    - This function knows how to interpret the binary data for its specific message type.

Using Unpackers

1. Message Identification:
    - When reading the binary file, the parser identifies the message type from the header.
2. Unpacking:
    - The appropriate unpacker is called with the message body.
    - The unpacker converts the binary data into a slice of Go interface{} types.
3. Data Interpretation:
    - The unpacked data is then associated with field names from the FMT message.
    - This creates a `DataFileMessage` object with accessible, typed data.

Main Parsing Loop

- The parser iterates through the file, reading each message header.
- Based on the message type in the header:
    1. It selects the appropriate unpacker.
    2. Applies the unpacker to the message body.
    3. Creates a `DataFileMessage` with the unpacked data.

GPS Interpolation

The `GPSInterpolated` struct is crucial for handling GPS data and timestamps:

- Purpose: To provide accurate timestamps for all messages, even those without explicit time information. For non-GPS messages, timestamps are interpolated based on their position relative to GPS messages.
- Functionality:
    1. Tracks GPS messages to establish a time base.
    2. Interpolates timestamps for non-GPS messages based on their sequence and known GPS times.
    3. Handles different time formats (week/milliseconds, UNIX time) used in various Mavlink implementations.
- Key Methods:
    - `FindTimeBase`: Establishes the initial time reference.
    - `GPSTimeToUnixTime`: Converts GPS time to UNIX timestamp.
    - `MessageArrived`: Processes each message to update timing information.
    - `SetMessageTimestamp`: Assigns an interpolated timestamp to non-GPS messages.

In-Depth Analysis of Binary Telemetry Data Parser

1. Binary File Structure

The binary file is structured as follows:

1. File Header (if any)
2. Sequence of Messages, each consisting of: a. Message Header (3 bytes)
   - Byte 1: 0xA3 (HEAD1)
   - Byte 2: 0x95 (HEAD2)
   - Byte 3: Message Type ID (0-255) b. Message Body (variable length)
3. Special FMT (Format) Messages:
   - Type ID: 128
   - Structure:
     - Type (1 byte)
     - Length (1 byte)
     - Name (4 bytes, null-terminated string)
     - Format (16 bytes, null-terminated string)
     - Columns (64 bytes, null-terminated, comma-separated string)

2. Initialization Process

2.1 Memory Mapping

- The entire binary file is memory-mapped using `mmap.MapRegion()`.
- This creates a byte slice (`dataMap`) that represents the file contents in memory.

2.2 Initial State Setup

- `offset`: Set to 0 (start of file)
- `remaining`: Set to file length
- `formats`: Empty map to store message formats
- `unpackers`: Empty map to store unpacking functions

NB:
Binary Navigation and Offset: Binary navigation is the process of moving through the binary file data, knowing exactly where each piece of information is located. The offset is a crucial concept in this navigation.

The offset is an integer that represents the current position in the binary data, measured in bytes from the start of the file. It's essentially a pointer to where the parser is currently reading in the file.

How the offset is determined and used:

1. Initial state: offset = 0 (start of file)
2. After reading each message: offset += message_length
3. When seeking a specific message: offset = known_message_start_position

For example, if you've just read a message that's 50 bytes long:

```
currentMessageLength := 50
reader.offset += currentMessageLength
```

The offset is critical because binary data doesn't have inherent separators like newlines in text files. By keeping track of the offset, the parser always knows where it is in the file and where the next piece of data begins.

3. First Pass: Format Discovery and Unpacker Creation

3.1 Scanning for FMT Messages

- The parser iterates through the file, looking for message headers.
- When a message with Type ID 128 (FMT) is found:

3.2 FMT Message Decoding

1. Read the FMT message body.
2. Unpack using the known FMT structure:
    - Type: 1 byte (uint8)
    - Length: 1 byte (uint8)
    - Name: 4 bytes (string)
    - Format: 16 bytes (string)
    - Columns: 64 bytes (string)

3.3 Creating DataFileFormat

For each FMT message:

1. Create a new `DataFileFormat` struct:

```
df := &DataFileFormat{
    Typ:     fmtType,
    Name:    name,
    Len:     length,
    Format:  format,
    Columns: strings.Split(columns, ","),
}
```

2. Parse the format string to set up:
    - `MessageStruct`: A string representation for binary.Read (e.g., "<BBnNZ")
    - `MessageTypes`: Slice of Go types corresponding to each format character
    - `MessageMults`: Slice of multipliers for scaled values

3.4 Dynamic Unpacker Creation

For each `DataFileFormat`:

1. Create an unpacker function:

```
unpacker := func(data []byte) ([]interface{}, error) {
    elements := make([]interface{}, 0)
    reader := bytes.NewReader(data)
    // For each format character:
    //   1. Read the appropriate number of bytes
    //   2. Convert to the correct Go type
    //   3. Apply any necessary scaling
    //   4. Append to elements slice
    return elements, nil
}
```

2. Store the unpacker in the `unpackers` map with the message type as the key.

4. Rewind and Second Pass: Data Extraction

4.1 Rewinding

- Reset `offset` to 0
- Reset `remaining` to file length
- Clear any temporary data structures

4.2 Main Parsing Loop

4.2.1 Reading Message Headers

1. Check if there are at least 3 bytes remaining:

```
if reader.remaining < 3 {
    return nil, io.EOF
}
```

2. Read the 3-byte header:

```
header := reader.dataMap[reader.offset : reader.offset+3]
```

3. Verify the header matches the expected pattern (0xA3 0x95):

```
if header[0] ≠ reader.HEAD1 || header[1] ≠ reader.HEAD2 {
    reader.offset++
    reader.remaining--
    continue // Keep searching for valid header
}
```

4. Extract the message type:

```
messageType := int(header[2])
```

4.2.2 Unpacker Dynamic Binding

The unpacker dynamically binds to the relevant data through several steps:

1. Message type identification: The message type is determined from the message header.
2. Unpacker selection: The appropriate unpacker is retrieved from the unpackers map using the message type as the key.
3. Data extraction: The message body is extracted based on the known message length for this type.
4. Unpacker application: The selected unpacker function is called with the message body as its argument:
```
elements, err := unpacker(messageBody)
```
5. Dynamic unpacking: Inside the unpacker function, it uses the format string associated with this message type to know how to interpret each byte:
```
for _, formatChar := range format.MessageFormats {
    switch formatChar {
    case 'f':
        value :=
math.Float32frombits(binary.LittleEndian.Uint32(data[offset:offset+4]))
        elements = append(elements, float64(value))
        offset += 4
    case 'I':
        value := binary.LittleEndian.Uint32(data[offset:offset+4])
        elements = append(elements, int(value))
        offset += 4
    // ... other cases for different format characters
    }
}
```

This dynamic binding allows the parser to adapt to different message structures without needing separate hard-coded unpacking logic for each message type.

4.2.3 Creating DataFileMessage

1. Create a new `DataFileMessage`:

```go
message := &DataFileMessage{
    Format:     format,
    Elements:   elements,
    FieldNames: format.Columns,
}
```

2. Process special messages (e.g., GPS for timing):

```go
if format.Name == "GPS" || format.Name == "GPS2" {
    reader.processGPSMessage(message)
}
```

4.2.4 Advancing to Next Message

1. Update the offset and remaining count:

```go
reader.offset += format.Len
reader.remaining = len(reader.dataMap) - reader.offset
```

5. GPS Time Interpolation

5.1 Establishing Time Base

The timebase is typically created from the first GPS message in the file, not necessarily the first record. Here's the process:

1. The parser scans through messages until it finds a GPS message.
2. From this GPS message, it extracts:
   - GPS Week number
   - Milliseconds into the week
3. These are converted to a UNIX timestamp:

```
func (clock *GPSInterpolated) GPSTimeToUnixTime(week int, msec int) float64 {
    epoch := SecondsInDay * (EpochDaysOffset*DaysInYear +
int((EpochLeapYearOffset-EpochStartYear)/YearsInLeapCycle) +
LeapYearAdjustment + EpochDaysFromYear - EpochDaysFromWeekday)
    return float64(epoch) + float64(SecondsInDay*DaysInWeek*week) +
float64(msec)*MillisecondsInSecond - LeapSecondsAdjustment
}
```

This timestamp becomes the initial timebase, and subsequent messages' times are calculated relative to this.

5.2 Interpolating Other Message Times

- For non-GPS messages:
  1. Calculate time since last GPS message based on message count and known rates
  2. Add this offset to the last known GPS time

6. Data Extraction and Geometry Creation

6.1 Filtering GPS Data

- As messages are processed, store relevant GPS data:

```
if message.Format.Name == "GPS" {
    lat, _ := message.GetAttribute("Lat")
    lon, _ := message.GetAttribute("Lng")
    // Store lat/lon if valid
}
```

6.2 Creating Geometry

- After parsing, convert stored GPS points to a geometry object:

```
coords := make([]float64, len(gpsData)*2)
for i, point := range gpsData {
    coords[i*2] = point.Lon
    coords[i*2+1] = point.Lat
}
sequence := geom.NewSequence(coords, geom.DimXY)
linestring := geom.NewLineString(sequence)
```

7. Key Aspects of Binary Parsing

7.1 Byte-Level Operations

- All data reading is done at the byte level:

```
value := binary.LittleEndian.Uint32(data[offset:offset+4])
```

7.2 Type Conversion

- Raw bytes are converted to appropriate Go types:

```
float32Value := math.Float32frombits(uint32Value)
```

7.3 Scaling

Scaling is used because some values in the binary data are stored as integers to save space, but they represent floating-point values in reality. This is most importantly used here for Latitude and Longitudinal values.
How scaling works:

1. The raw integer value is read from the binary data.
2. This integer is then multiplied (or divided) by a scaling factor to get the true value.

For example, GPS coordinates are often stored as integers that need to be divided by 10^7 to get the actual decimal degrees:

```
rawLat := int32(binary.LittleEndian.Uint32(data[offset:offset+4]))
actualLat := float64(rawLat) * 1e-7
```

In this case, 1e-7 is the scaling factor. This allows storing a latitude like 37.7749° as the integer 377749000, saving space while maintaining precision.

7.4 String Handling

- Strings are often null-terminated and fixed-length:

```
nullIndex := bytes.IndexByte(data, 0)
if nullIndex != -1 {
    data = data[:nullIndex]
}
stringValue := string(data)
```

8. Ensuring Completeness of Unpackers

- The parser ensures all message types have unpackers by:
    1. Keeping a set of encountered message types during the first pass.
    2. Verifying that each type in this set has a corresponding unpacker.
    3. If any are missing, it either creates a generic unpacker or logs an error.

9. Binary Navigation

Advancing to Next Message: Advancing to the next message involves several steps:

1. Update the offset:
```
reader.offset += currentMessageLength
```
2. Update the remaining data count:
```
reader.remaining = len(reader.dataMap) - reader.offset
```
3. Check if there's enough data for another message header:
```
if reader.remaining < 3 {
    return nil, io.EOF // End of file reached
}
```
4. Read the next message header:
```
header := reader.dataMap[reader.offset : reader.offset+3]
```
5. Validate the header and extract the message type:
```
if header[0] == reader.HEAD1 && header[1] == reader.HEAD2 {
    messageType := int(header[2])
    // Process the message...
} else {
    // Invalid header, try to resynchronize
    reader.offset++
    reader.remaining--
    continue
}
```

This process ensures that the parser moves accurately from one message to the next, handling any potential synchronization issues.