

Telemetry Processing: From Bytes to Structs

Practical 03 • 27 October 2025

Instructor: Dr. Stuart James (stuart.a.james@durham.ac.uk)

Scenario

After a long day in the field, your team's autonomous rover finally transmits its telemetry back to mission control. The rover itself is far away and inaccessible, so all you receive are fragments of its activity: a stream of text-based status messages captured from its serial output, a compact binary "black box" file of raw sensor readings, and a short preflight checklist completed before launch. Your role is to reconstruct what happened by designing small C programs that read input from the terminal and files, store related data using structs, and reinterpret binary sensor bytes with a union. By the end of the session, you will visualise how these elements interact through a diagram that illustrates the flow of data from the rover's logs and memory into meaningful, structured telemetry records.

Learning Objectives

- Read input from the terminal (`stdin`) and command-line arguments.
- Write structured data to files using `fopen`, `fprintf`, `fclose`.
- Model related data using `struct` and understand memory grouping.
- Use `union` to interpret the same bytes as different data types.
- Communicate a systems-level concept visually through a data flow or memory layout diagram.

Prerequisites

- Access to Linux, macOS, or `mira.dur.ac.uk`.
- Confidence compiling small C programs with `gcc` / `Make`.
- Awareness of how command-line arguments (`argv`) and file I/O work.

Tasks

Task 1: Filtering live stream to file

Solo**Intro****45m**

Outcome: Learn to combine terminal input, arguments, and file output — a core skill for working with live telemetry data.

► Activity

In real telemetry systems, data rarely arrives as a tidy file. Instead, it streams continuously from sensors or communication links, and engineers must extract only the relevant messages for analysis. This task introduces the same idea on a smaller scale.

You will write a tool called `filterlog.c` that simulates a ground-station log filter. It should:

- take two arguments: a keyword (e.g. `ERROR`) and an output filename,
- read lines of text from `stdin` (simulating the live telemetry feed),
- write only lines containing that keyword into the specified file (simulating a filtered event log),
- compile using a simple `Makefile`.

By redirecting input with a pipe (`|`) or file redirection, you can mimic telemetry playback from stored logs, just as mission engineers would when replaying captured data streams.

You can use the include `filterlog_data.txt` to test your program.

You can run your tool as:

```
./filterlog ERROR errors.txt < filterlog_data.txt  
cat errors.txt
```

► Activity

Extend your program so that it can also read directly from a file instead of only from `stdin`. Add a third optional argument to specify an input file. If this argument is provided, open it using `fopen` for reading; otherwise, continue reading from `stdin` as before.

Example usage:

```
# Process a saved telemetry file directly  
./filterlog ERROR errors.txt filterlog_data.txt
```

This change reflects how real telemetry analysis tools support both *live mode* (connected to a stream) and *replay mode* (reading from stored data logs).

◊ Reflection

Why does the program need both `stdin` and `argv`? When might redirecting input (`|`) be useful when analysing real telemetry streams or logs?

Task 2: Structs and unions for telemetry

SOLO**INTERMEDIATE****45m**

Outcome: Represent and interpret telemetry values using struct and union — mirroring how real systems encode and decode sensor data.

► Activity

Telemetry data represents the internal state of a system — for example, temperature readings, battery levels, or GPS status — that must be transmitted efficiently to a ground station. To organise these values clearly in code, engineers group related fields together in a struct, allowing them to pass a complete telemetry packet as a single unit. However, on the transmission side, these same values are often compacted into raw bytes for efficiency, and decoding them back into meaningful values requires interpreting those bytes correctly — a task where union becomes extremely useful.

In this exercise, you will create a program `telemetry.c` that models this process:

- define a struct holding three fields: temperature (float), battery (int), and GPS lock state (int);
- create a union that stores either four individual bytes or a single float;
- assign example values, print the telemetry packet in human-readable form, and then show how the same four bytes can be interpreted as a floating-point sensor value.

This mirrors how real telemetry pipelines convert between structured engineering data and the compact binary packets transmitted across networks or radio links.

Task 3: Assessed: Diagramming the data flow

ASSESSED**HARD****45m**

Outcome: Demonstrate conceptual understanding by illustrating the program's data flow and memory layout.

Scenario: A new team member has joined and asks: “How does this telemetry program actually move data from the terminal into memory, through the struct and union, and finally out to a file?”

Your task is to **draw a clear diagram** that answers this.

► Activity

Create a half-page hand-drawn diagram saved as dataflow.pdf or dataflow.png that shows:

- The flow of data from terminal input → program arguments → struct Telemetry → union FloatBytes → file output.
- The relationship between variables in memory (label each field and its approximate size).
- The distinction between text I/O (lines of characters) and binary I/O (raw bytes).
- A short caption (3–5 sentences) explaining your design choices and how the C features (struct, union, file I/O) work together.

Important

This assessed task focuses on *conceptual clarity*, not artistic skill. Diagrams may be neatly hand-drawn (scanned/photo). The goal is to explain, visually, how the system works.

Submission Checklist

Submission Checklist

- filterlog.c (Task 1)
- telemetry.c (Task 2)
- dataflow.pdf or dataflow.png (Task 3 — assessed diagram)
- Optional: diagram_caption.txt if you prefer to write the description separately.

Place all the above files in a folder named Practical3 and submit via the coursework system.

```
Practical1/  
|-- [Intro scripts]  
Practical2/  
|-- [Makefiles + NCC batch script]  
Practical3/  
|-- filterlog.c  
|-- telemetry.c  
|-- dataflow.png  
|-- diagram_caption.txt  
Practical4/  
|-- ...
```