# Documentation

## for `main.rs`

Rust ASN.1 DER Decoder → JSONL

Beginner-friendly, line-by-line explanation with schema-based decoding.

**DER** **ASN.1** **JSONL**

January 6, 2026

# Contents

# 1 What this program does (high level)

This program:

1. Reads an ASN.1 schema file (a text file describing data structures).
2. Reads one or more input files containing DER-encoded ASN.1 data (binary).
3. Parses the binary data as TLV records (TLV = Tag + Length + Value).
4. Uses the schema to interpret those TLVs as structured fields (CHOICE / SEQUENCE / SET / primitives).
5. Outputs one JSON object per record (JSONL format: JSON Lines), where:
   - Keys are field names from the schema
   - Values are written as **hex strings** (not decimal numbers)

So the output is:

> decode DER file(s) → produce `.jsonl` files with schema-based JSON.

# 2 How to read this document

- The explanations assume you have **no prior Rust knowledge**.
- I refer to the approximate line ranges in `main.rs`.
- For each function, you will see:
  - Purpose
  - Inputs
  - Outputs
  - How it works internally
  - Line-by-line explanation (grouping only when many lines do the exact same simple thing)

# 3 Imports (lines 1–11)

```rust
use anyhow::{anyhow, Context, Result};
use clap::Parser;
use memmap2::Mmap;
use rayon::prelude::*;
use regex::Regex;
use std::collections::{HashMap, HashSet};
use std::fs::File;
use std::io::{BufWriter, Write};
use std::path::{Path, PathBuf};
use std::time::Instant;
use walkdir::WalkDir;
```

### Line-by-line explanation

- **Line 1:** Imports the `anyhow` error helpers:
  - `Result` = a convenient "success or error" return type
  - `Context` = adds extra message to errors
  - `anyhow!` = creates an error with a message
- **Line 2:** Imports `clap::Parser` so command-line options can be parsed into a struct.
- **Line 3:** Imports `Mmap` (memory mapping) for fast file reading without copying everything.
- **Line 4:** Imports Rayon helpers for parallel processing.
- **Line 5:** Imports `Regex` to parse schema text.
- **Line 6:** Imports hash maps/sets for fast lookups.
- **Line 7:** Imports `File` for opening/creating files.
- **Line 8:** Imports buffered writing and the `Write` trait.
- **Line 9:** Imports path types for file paths.
- **Line 10:** Imports `Instant` for timing the program.
- **Line 11:** Imports `WalkDir` for scanning directories recursively.

## 4 Command-line arguments: `Cli` (lines 13–41)

### Purpose

Defines what arguments the user can pass on the command line.

### Key code

```
/// CLI arguments
#[derive(Parser, Debug)]
#[command(
    author,
    version,
    about = "Ultra-fast ASN.1 DER Decoder -> JSONL (schema-based, hex-only values, no decimal conversion)",
    long_about = None
)]
struct Cli {
    /// ASN.1 schema file
    #[arg(long = "schema")]
    schema: PathBuf,

    /// Root ASN.1 type (e.g. PGWRecord). Use "auto" to infer.
    #[arg(long = "root-type")]
    root_type: String,

    /// Output directory
    #[arg(long = "output-dir")]
    output_dir: PathBuf,

```

```
    /// Optional: only decode files matching these extensions (comma-separated), e.g. "dat,bin"
    #[arg(long = "ext")]
    ext: Option<String>,

    /// DER-encoded input files or directories (directories scanned recursively)
    #[arg(required = true)]
    inputs: Vec<PathBuf>,
}
```

### Beginner explanation

- This struct is a **template** for all command-line settings.
- The program uses Clap to automatically fill this struct from the user's command.
- Example flags:
  - `-schema path/to/schema.asn1`
  - `-root-type PGWRecord` (or `auto`)
  - `-output-dir out/`
  - `-ext dat,bin`
  - and then input files or folders at the end

# 5 Schema data structures

## 5.1 `FieldSpec` (lines 43–51)

### Purpose

Represents a single field inside a `SEQUENCE` or `SET`.

### Code

```
#[derive(Debug, Clone)]
struct FieldSpec {
    name: String,
    field_type: String,
    #[allow(dead_code)]
    optional: bool,
    is_sequence_of: bool,
}
```

### Explanation

- `name`: the field name (becomes a JSON key).
- `field_type`: the ASN.1 type name for this field.
- `optional`: whether the schema says it is optional.
- `is_sequence_of`: whether it is a list (`SEQUENCE OF X`).

### 5.2 `Asn1Schema` (lines 53–61)

#### Purpose

Stores parsed schema information in fast lookup tables.

#### Code

```
#[derive(Debug, Default)]
struct Asn1Schema {
    choices: HashMap<String, HashMap<u32, (String, String)>>,
    sequences: HashMap<String, HashMap<u32, FieldSpec>>,
    sets: HashMap<String, HashMap<u32, FieldSpec>>,
    primitives: HashMap<String, String>, // type_name -> primitive kind
    aliases: HashMap<String, String>,
}
```

#### Explanation

- `choices`: CHOICE type name → (tag → (field name, field type)).
- `sequences`: SEQUENCE type name → (tag → field spec).
- `sets`: SET type name → (tag → field spec).
- `primitives`: type name → primitive kind (like `INTEGER`, `OCTET STRING`).
- `aliases`: type name → another type name (renaming / aliasing).

## 6 Synthetic CHOICE tags (lines 63–67)

### Why this exists

Some CHOICE alternatives may not have explicit tags. The program creates **synthetic (fake)** internal tags so it can still store these alternatives consistently.

```
const SYNTH_CHOICE_BASE: u32 = 0xFFFF_FF00;

#[inline]
fn is_synth_choice_tag(t: u32) -> bool {
    t >= SYNTH_CHOICE_BASE
}
```

#### Explanation

- `SYNTH_CHOICE_BASE` is the starting range for these fake tags.
- `is_synth_choice_tag` checks if a tag number is in that synthetic range.

## 7 Parsing the schema: `impl Asn1Schema`

### 7.1 `Asn1Schema::parse` (starts around line 70)

#### Purpose

Reads schema text and fills in: `choices`, `sequences`, `sets`, `primitives`, and `aliases`.

#### Inputs

- `schema_text: &str` — the entire schema file as a string.

#### Outputs

- `Result<Self>` — returns a built schema or an error.

#### How it works internally (simple)

1. Create regular expressions for ASN.1 patterns.
2. Remove comments.
3. Find type definitions like `TypeName ::= SEQUENCE {...}` or `CHOICE {...}`.
4. Store the discovered rules into maps for quick decoding later.
5. Also store type aliases (`A ::= B`).

### 7.2 `Asn1Schema::resolve_alias` (line 192)

#### Purpose

Follow alias chains like `A ::= B` and `B ::= C`, so `A` resolves to `C`.

#### Inputs

- `t: &str` — a type name.

#### Outputs

- `&str` — the final resolved type name.

#### How it operates

It looks up `t` in the alias map repeatedly (up to 16 steps to avoid infinite loops).

### 7.3 `Asn1Schema::knows_type` (line 204)

#### Purpose

Checks whether a type name exists in the schema (after alias resolving).

#### Inputs

- `t: &str`

#### Outputs

- `bool`

#### How it operates

Resolves aliases first, then checks if the type is in any known category: CHOICE, SEQUENCE, SET, or primitive.

# 8 TLV representation: `Tlv` (lines 213–223)

### Purpose

Represents one parsed DER object (one TLV).

```
struct Tlv<'a> {
    tag_class: u8,
    constructed: bool,
    tag_num: u32,
    length: usize,
    value: &'a [u8],
    raw: &'a [u8],
}
```

### Explanation

- `tag_class`: what category the tag belongs to (universal/application/context/private).
- `constructed`: true means the value contains nested TLVs.
- `tag_num`: the numeric tag identifier.
- `length`: how many bytes the value has.
- `value`: a slice pointing to the value bytes.
- `raw`: a slice pointing to the full TLV bytes (tag+length+value), useful for raw hex output.

# 9 JSON writing helpers

## 9.1 `write_json_key` (line 226)

### Purpose

Write a JSON key safely (with minimal escaping).

### Inputs

- `w: &mut W` — an output writer
- `key: &str` — the key text

### Outputs

- `Result<()>`

### How it works internally

Writes `"key"` and escapes characters like quotes and backslashes so the JSON remains valid.

## 9.2 `hex_encode_into` and `write_hex_json` (lines 251 and 268)

### Purpose

Convert raw bytes to lowercase hex and write them as JSON strings.

#### Why a scratch buffer is used

The program reuses a buffer so it does not allocate a new string for every value. This is much faster for large files.

### 9.3 `lower_first` (line 277)

#### Purpose

Lowercases the first letter of a string (often used for JSON keys).

#### Inputs

- `s: &str`

#### Outputs

- `String`

## 10 The decoder: `DerDecoder` (starts line 285)

### Purpose

Holds the schema and provides methods to:

- parse TLVs from bytes
- interpret them using schema types
- write JSON output

```
struct DerDecoder {
    schema: Asn1Schema,
    cs_choice_index: HashMap<u32, String>,
}
```

### Explanation

- `schema` contains all parsed ASN.1 information.
- `cs_choice_index` maps synthetic CHOICE tags to CHOICE type names for faster “auto” decoding.

## 11 Decoder functions (the heart of the program)

### 11.1 `DerDecoder::new` (line 291)

#### Purpose

Builds the decoder and constructs the synthetic CHOICE index.

#### Inputs

- `schema: Asn1Schema`

#### Outputs

- `DerDecoder`

### 11.2 `DerDecoder::parse_tlv` (line 305)

#### Purpose

Reads one TLV from a byte slice at a specific offset.

#### Inputs

- `data: &[u8]` — the full file bytes
- `offset: usize` — starting position

#### Outputs

- `Option<(Tlv, usize)>`
  - `Some((tlv, new_offset))` if parsing succeeds
  - `None` if parsing fails or reaches the end

#### How it works internally (simple)

1. Read tag information (class, constructed flag, tag number).
2. Read the length (short or long form).
3. Slice out the value bytes and raw bytes.
4. Return the TLV and the next offset.

#### Why so many bounds checks?

Binary parsing can break badly if lengths are wrong. Bounds checks prevent reading past the end of the file and keep decoding safe.

### 11.3 `DerDecoder::write_type` (line 380)

#### Purpose

Given raw bytes and a schema type name, writes the correct JSON representation.

#### Inputs

- `data: &[u8]` — bytes to decode
- `type_name: &str` — schema type name
- `out: &mut W` — output writer
- `scratch: &mut Vec<u8>` — reusable hex buffer

#### Outputs

- `Result<()>`

**How it works internally**

1. Resolve aliases.
2. If primitive: write hex string.
3. If SEQUENCE/SET: parse nested TLVs and write a JSON object.
4. If CHOICE: select the correct alternative and write the chosen value.

## 11.4 `DerDecoder::write_sequence` (line 401)

**Purpose**

Decodes a constructed value as a JSON object using SEQUENCE/SET field definitions.

**Inputs**

Same general pattern: `data`, `type_name`, `out`, `scratch`.

**Outputs**

`Result<()>`

**How it works internally**

- Write `{`.
- Loop: parse nested TLVs.
- For each TLV, look up its tag number in the schema and write `"field": value`.
- Write `}`.

## 11.5 `DerDecoder::write_sequence_of` (line 460)

**Purpose**

Decodes `SEQUENCE OF X` as a JSON array.

**How it works internally**

Write `[`, decode each element, separate with commas, then write `]`.

## 11.6 `DerDecoder::choice_alt_matches_tlv` (line 494)

**Purpose**

Checks whether a TLV appears compatible with a CHOICE alternative type.

**Inputs**

- `alt_type: &str`
- `tlv: &Tlv`

**Outputs**

- `bool`

#### How it works internally

Resolves aliases and then compares TLV properties (tag number, constructed flag, etc.) to what that alternative expects.

### 11.7 `DerDecoder::write_choice` (line 513)

#### Purpose

Decodes a CHOICE value by selecting which alternative matches.

#### Inputs

- `tlv: &Tlv`
- `choice_name: &str`
- `out`, `scratch`

#### Outputs

`Result<()>`

#### How it works internally

1. Look up all alternatives for the CHOICE.
2. If the TLV tag matches a tagged alternative, use it directly.
3. Otherwise test alternatives using `choice_alt_matches_tlv`.
4. Output JSON like `{"chosenName": value}`.
5. If nothing matches, output the raw hex as an "unknown" alternative.

### 11.8 `DerDecoder::write_root_tlv_with_type` (line 583)

#### Purpose

Decode a top-level TLV using a known root type and write JSON.

#### Inputs

- `tlv: &Tlv`
- `root_type: &str`
- `out`, `scratch`

#### Outputs

`Result<()>`

### 11.9 `DerDecoder::write_auto_record` (line 599)

#### Purpose

Decode a top-level TLV without a specified root type ("auto mode").

#### Inputs

- `tlv: &Tlv`
- `out, scratch`

#### Outputs

`Result<()>`

#### How it works internally (typical goal)

Try to infer the best type (often using synthetic CHOICE indexing). If it cannot infer, still output something useful (like tag info and raw hex) so data is not lost.

## 12 File discovery helpers

### 12.1 `expand_inputs` (line 622)

#### Purpose

Converts a list of input files/folders into a flat list of files to decode.

#### Inputs

- `inputs: &[PathBuf]`
- `allowed_exts: Option<&HashSet<String>>`

#### Outputs

`Result<Vec<PathBuf>>`

#### How it works internally

- If an input is a file, include it if it passes extension filtering.
- If it is a directory, recursively scan it with `WalkDir`.
- Otherwise return an error.

### 12.2 `should_include` (line 649)

#### Purpose

Applies optional extension filtering.

#### Inputs

- `path: &Path`
- `allowed_exts: Option<&HashSet<String>>`

#### Outputs

`bool`

### How it works internally

- If no extensions were specified, include everything.
- Otherwise compare the file extension (lowercased) to the allowed set.

## 13 Decoding one file: `process_file` (line 661)

### Purpose

Decodes one input file into one output `.jsonl` file.

### Inputs

- `decoder: &DerDecoder`
- `root_type: &str`
- `in_path: &Path`
- `out_dir: &Path`

### Outputs

- `Result<usize>` = number of decoded records in that file

### How it works internally (simple)

1. Open the file.
2. Memory-map it for fast reading.
3. Create an output file `<inputname>.jsonl`.
4. Use a large buffered writer for speed.
5. Loop over the file:
    - parse one TLV
    - decode it to JSON (auto mode or root-type mode)
    - write it and add a newline
6. Flush and return the record count.

## 14 Program entry point: `main` (line 714)

### Purpose

Connects everything: read arguments → read schema → find files → decode files.

### Outputs

`Result<()>` (success or error)

### Overall step-by-step flow

1. Parse command-line arguments into `Cli`.
2. Start a timer.
3. Parse optional extension filtering.
4. Read schema file and build `Asn1Schema`.
5. Build `DerDecoder`.
6. Create the output directory if missing.
7. Determine decoding mode (explicit root type or auto).
8. Expand inputs into a list of files.
9. Process files (often in parallel using Rayon).
10. Print totals (records and elapsed time).

## 15 How the functions work together (big picture)

### Data flow through the program

1. **`main`** reads arguments and loads the schema.
2. **`Asn1Schema::parse`** turns schema text into lookup maps.
3. **`DerDecoder::new`** prepares helper indexes for decoding.
4. **`expand_inputs`** finds every file to decode.
5. For each file, **`process_file`**:
   - memory-maps the file bytes
   - repeatedly calls **`parse_tlv`** to extract each record
   - uses **`write_auto_record`** or **`write_root_tlv_with_type`**
   - which delegates to **`write_type`**, **`write_sequence`**, **`write_choice`**, etc.
   - and finally writes JSON lines to the output file

## 16 How the final output is generated

- Each top-level TLV becomes exactly **one JSON object**.
- That JSON object is written to the output `.jsonl` file.
- A newline is written after it.
- This is why it is called **JSON Lines**: one record per line.