ChatGPT

Current Coverage

The MIDI 2.0 specification is expansive – it spans multiple documents covering different facets (Capability Inquiry, Profiles, Property Exchange, UMP/Protocol, etc.) [1] [2]. The current `midi2-reader` implementation appears to parse only a subset of this breadth. Based on the codebase, it seems focused primarily on core MIDI 2.0 message definitions (e.g. the **Universal MIDI Packet (UMP) format and MIDI 2.0 Protocol** messages) and perhaps parts of **MIDI-CI (Capability Inquiry)**. Key elements like the new 32-bit **Channel Voice messages** and fundamental UMP structure are likely represented. However, large portions of the spec do not yet seem to be machine-modeled. For example, **Profile configuration rules** and **Property Exchange resources** (defined in separate spec documents) are probably missing or only stubbed. The **MIDI 2.0 Overview** narrative and explanatory sections are also not parsed (which is acceptable, as they are mostly descriptive text). In summary, the current coverage is partial – it captures some message formats and fields, but **many sections of the official spec are not yet parsed into the tool's data model**. This means the repository is far from a one-stop, all-encompassing source of MIDI 2.0 spec details.

From what is implemented, the extracted spec content is organized in code, but its structure may be ad-hoc. For instance, message formats might be hard-coded or parsed into custom classes/structures. There is evidence of basic parsing of tables (e.g. for UMP message types and their fields) and some constants. But a **unified, comprehensive schema** of the entire spec is not present. The clarity of the extracted data suffers accordingly – one can't easily ask "what does message X contain?" without reading code or partial outputs. **Accessibility** is limited: if an external program or an LLM needs info (say the bit layout of a Note On message), it may have to rely on `midi2-reader`'s internals or logs rather than a clean data file. In short, **only portions of the MIDI 2.0 spec are modeled**, and those that are modeled are not yet presented in a clean, self-contained format for easy consumption.

LLM Compatibility Analysis

The goal is to support large language models in generating Swift implementations of MIDI 2.0 features. From that perspective, the current format of the parsed spec content is **not ideal for LLM consumption**. LLMs work well with structured, explicit information (or well-formatted documentation) provided in their context. If `midi2-reader` is just outputting raw text or using internal data structures, an LLM will struggle to **extract the precise details** needed for code generation. For example, an LLM asked to "implement a Swift struct for the MIDI 2.0 Note On message" would benefit from a concise definition: something like a schema of the Note On message ("fields: note number (7-bit), velocity (16-bit), etc."). If instead the info is buried in paragraphs of spec text or scattered in code, the model might miss details or make mistakes.

Structure and Clarity: The existing representation likely lacks a standardized structure (such as JSON or a formal schema). This can confuse an LLM. A structured format (key-value pairs or tables for each message and field) would let the LLM reliably identify each component. Currently, the content might be semi-structured (perhaps some markdown or code comments), which is only semi-readable to a model. Additionally, any inconsistencies in naming (e.g. using different terms for the same concept across the data) would further reduce LLM comprehension. Without uniform naming and formatting, the model could produce inconsistent Swift code (for instance, misnaming a field or misinterpreting a value's range).

Swift-Specific Considerations: The current output doesn't seem tailored to Swift language conventions. An LLM might need to know how spec concepts map to Swift types or enums. For instance, MIDI 2.0 defines many **enumerations and bitfields** (like message type codes, controller numbers, etc.), and Swift implementations would ideally use enums/structs for these. If the spec data is not organized in a way that highlights "this is an enum of possible values" or "this is a 4-bit field for group", the LLM might not infer the best Swift representation. The format should ideally hint at types (e.g. integer size, value ranges) so the LLM can choose `UInt4`, `UInt7` (7-bit represented in a larger type), etc., in Swift. Right now, such type cues are probably absent or implicit.

In summary, the current data format is **only moderately suitable** for LLM consumption. It provides some useful info but not in a readily digestible, structured way. This raises the risk of the LLM producing incorrect or incomplete Swift code. To maximize fidelity in code generation, the spec needs to be **presented in a more structured, explicit format** that aligns with coding needs.

Critical Gaps

Several gaps and issues are evident in the current implementation that threaten spec fidelity and completeness:

- **Incomplete Spec Coverage:** As noted, whole areas of the MIDI 2.0 spec are missing. Notably, **Property Exchange** (the mechanism for querying/setting device properties) and **Profile configuration** rules are likely unparsed. These parts contain lists of standard property IDs, JSON data schemas, etc., which are crucial for a full implementation. Similarly, the **MIDI 2.0 File/Clip format** (for SMF compatibility) might be absent. This means an LLM won't even have access to those feature definitions, leading to holes in generated code (for example, no support for Property Exchange messages because the model never saw their format).

- **Inconsistencies and Naming Issues:** If the parsing was done via scraping text, some names or values might not exactly match the spec. For instance, a spec table might label a field "Note Number (7-bit)", but the code might capture it as `note_number` somewhere and "Note" elsewhere. Inconsistent naming or formatting across different parsed sections can confuse an LLM. Moreover, certain messages could be modeled at differing detail levels (some with every bit field labeled, others perhaps lumping bits together) – such inconsistency reduces the overall fidelity. The lack of a consistent naming convention (e.g. always using TitleCase vs snake_case, etc.) in the extracted data is a gap to address for machine-readability.

- **Brittle Parsing Strategies:** The approach used to parse the spec seems fragile. Likely, it relies on pattern matching in text or specific PDF layout assumptions. Spec documents often include complex tables, multi-column layouts, and footnotes. A brittle parser might mis-read a table split across pages, or break if a line wraps differently. For example, a regex looking for "Message Name – Byte 1 – Byte 2…" could fail if formatting changes or if a new message type doesn't conform to the assumed pattern. This brittleness can result in **missing fields or incorrect values** in the output without obvious errors. Thus, the current pipeline might not guarantee 100% fidelity to the source documents – some fields might be dropped or mis-assigned. Any such errors propagate to the LLM input, causing it to generate wrong code (garbled constants, wrong bit masks, etc.).

- **Lack of Metadata and Cross-Referencing:** The extracted content likely omits metadata like *which official document and section a piece of data came from*, or *version information*. The MIDI 2.0 spec has already seen revisions (e.g. 1.0 vs 1.1 of certain docs [3]). If the tool's data doesn't track

spec version or document references, it may be hard to update or verify. An LLM also can't verify consistency across sections (e.g. ensuring that a defined constant matches its usage elsewhere) if cross-references aren't preserved. This gap means the current data might become outdated or inconsistent as the spec evolves, and an LLM wouldn't be aware of updates.

Overall, these gaps – missing entire spec areas, inconsistent representation, fragile parsing, and lack of context metadata – **reduce the fidelity and reliability** of `midi2-reader`'s output. In its current state, it cannot claim to offer a complete, authoritative machine-readable MIDI 2.0 spec, which is the ultimate goal.

Recommendations

To evolve the project into a **canonical, machine-perfect MIDI 2.0 spec representation tool**, a series of improvements is recommended across parsing, data format, and content organization. Below are detailed suggestions:

Robust Parsing Strategies for Full Spec Fidelity

Rather than ad-hoc text scraping, the project should adopt a more robust parsing pipeline. This might involve multiple stages:

- **Structured PDF Parsing:** Use PDF processing libraries to extract content with layout awareness (for example, detect tables and headings properly). For instance, tables of message formats should be captured cell by cell, rather than relying on regex across raw text which can break [4] [5]. Specialized parsing for known structures (message definition tables, lists of defined values) will improve accuracy. If available, use the official HTML versions of the spec or convert PDF to HTML/XML to preserve structure.

- **Heuristic and Manual Hybrid Approach:** Some spec sections might defy perfect automated parsing (e.g. narrative text or diagrams). For these, provide **manual annotations** or extraction rules. For example, if the Property Exchange spec lists resources in a paragraph form, it might be easier to manually format that once, then include it as structured data. The pipeline can incorporate such curated data to fill gaps that automation misses.

- **Validation of Parsed Data:** Incorporate cross-checks to catch errors. For instance, the number of defined message types in the UMP spec is known (the spec enumerates all statuses in a table [6] [4]). If the parser's output count diverges, that's a red flag. Similarly, implement sanity checks (like field widths summing to the total bits of a message, numeric ranges matching expected bit-size limits, etc.). This will catch parsing mistakes or omissions early.

By improving parsing in these ways, the tool can more reliably achieve **100% fidelity** to the official spec content, which is essential for downstream use by LLMs.

Adopt a Structured, LLM-Friendly Data Format

Transition the output to a **machine-readable format** that is both rigorous and easy for LLMs to ingest. Possible formats include JSON, YAML, or even a Python dictionary exported to a JSON – anything that clearly represents the spec as data rather than free text. For example, define a JSON schema for MIDI 2.0 messages:

```
{
  "messages": [
    {
      "name": "Note On",
      "type": "Channel Voice Message (MIDI 2.0)",
      "statusCode": "0x90",
      "fields": [
        {"name": "Group", "bits": 4, "description": "UMP Group (0-15)"},
        {"name": "Channel", "bits": 4, "description": "Channel within group
(0-15)"},
        {"name": "Note Number", "bits": 7, "description": "Note index
(0-127)"},
        {"name": "Velocity", "bits": 16, "description": "High-resolution
velocity value"}
      ]
    },
    ...
  ]
}
```

This is just an illustrative snippet, but organizing data this way has huge benefits. An LLM can **explicitly see each field name and size**, making it straightforward to generate a Swift struct or class with matching properties. JSON/YAML is naturally consumed by many tools and can be pretty-printed or queried. It's also unambiguous – unlike free text, there's little room for misinterpretation of a numeric field width or name.

One could also consider **Protobuf or XML**, but JSON/YAML has the advantage of being very LLM-friendly (LLMs have been well-trained on JSON patterns). Moreover, JSON can handle nested structures for things like **enumerations**: e.g., for Registered Controller values or Bank Select messages, we can include sub-objects listing allowed values or meaning of each code.

If readability for humans is also desired, an **annotated Markdown** could accompany the JSON – e.g., auto-generate tables in Markdown from the JSON data. However, the primary source of truth should be a structured dataset. The LLM can be given sections of that dataset as needed, focusing its attention on relevant parts (for instance, just feed the JSON for the message type that the LLM needs to implement in code).

In summary, moving to a structured format will make the spec **truly machine-readable** and thereby LLM-ready. It reduces guesswork and allows programmatic access (so even beyond LLMs, any developer could use the JSON to generate code in various languages).

Enhanced Content Structuring, Naming, and Metadata

To maximize utility, the content itself should be restructured with consistency and rich annotations:

- **Consistent Naming Conventions:** Decide on a naming scheme for all entities and stick to it. For example, use `UpperCamelCase` for message names ("NoteOn" rather than "Note on"), and perhaps `snake_case` or a consistent style for field keys. Align these with typical Swift naming where possible (Swift uses CamelCase for types, lowerCamel for properties). For instance, a JSON

field `"noteNumber"` is immediately usable as a Swift property name. Consistency here will prevent the LLM from having to guess or normalize names, reducing errors.

- **Include Spec Metadata:** Each item of data should carry provenance and context. For example, tag each message or feature with the spec document and section identifier it came from (e.g., "Defined in *M2-104 §7.2.3*"). Also record the spec version (v1.0, v1.1, etc.) used. This not only future-proofs the dataset (when specs update, it's easier to know what needs changing) but can also be used in prompts to ground the LLM ("according to MIDI 2.0 spec v1.1, section X…").

- **Add Descriptive Annotations:** Besides raw field definitions, include short descriptions and notes from the spec. For instance, if a field has special meaning (like the 16-bit velocity essentially combines MSB/LSB of MIDI 1.0 velocity), include that note. These annotations help the LLM understand *semantics*, not just the bits. The Swift code it generates can then include comments or choose appropriate data types. Another example: for Enum-like fields (say a "Per-Note Management" message has an 8-bit field where certain values mean on/off), listing those values with names in the data would let the LLM create a Swift `enum` with cases for each. Without such detail, the LLM might just use a numeric type and lose semantic clarity.

- **Organize by Categories:** It might help to break the spec data into logical groupings in the structure. E.g., have separate sections for "Channel Voice Messages," "System Messages," "Utility Messages," "Discovery/CI Messages," "Property Exchange," etc., mirroring how the spec itself is organized [4] [7]. This way, an LLM or developer can focus on one category at a time. It also helps ensure no category is forgotten. The current code likely doesn't clearly separate these (or omits some categories entirely); a reorganization will make gaps obvious and navigation easier.

By refining the content structure and naming, we ensure that the machine-readable spec is not just complete, but also **coherent and self-explanatory**. This greatly aids LLMs in using the data effectively – leading to more accurate and consistent Swift code generation that mirrors the official spec terminology.

Roadmap to a Canonical MIDI 2.0 Spec Tool

Achieving 100% machine-readability and completeness will be an iterative process. Here is a suggested roadmap to incrementally evolve `midi2-reader` into a authoritative spec representation:

1. **Audit and Complete Spec Coverage:** First, enumerate all sections of the MIDI 2.0 spec and map them against the current implementation. Fill in the missing pieces one by one. For example, if "Property Exchange" is missing, implement parsing or manual data entry for its resource definitions and messages. Do the same for Profiles, CI details, and the MIDI 2.0 File format. Use the official spec documents as a checklist to ensure nothing is skipped. The Art+Logic summary of the five core documents can serve as a high-level checklist [1] [2]. By the end of this step, the tool should encompass *all* MIDI 2.0 features defined in those documents.

2. **Refactor into a Structured Data Model:** As the data comes in, refactor the code to build the structured format (as discussed above) rather than, say, printing text. Introduce data classes or dictionaries that accumulate spec info in-memory, then have an export function to JSON/Markdown. This refactoring will enforce consistency (since you'll define one schema for all messages, etc.) and make the output generation straightforward. It also separates data from presentation, which is good for maintainability.

3. **Implement Versioning and Testing:** Lock in the spec version that the data represents (initially MIDI 2.0 v1.0 or v1.1). Write unit tests or validation scripts that, for example, cross-verify known values. Perhaps compare against another source – e.g., the Rust `midi2` crate claims to cover every message [8]; you could write a small script to ensure every message that appears in that crate also exists in your JSON (not to copy it, but to validate completeness). Additionally, test that JSON output can be loaded and interpreted by a simple program (simulating how an LLM or any consumer would use it).

4. **Enhance Documentation and Accessibility:** Provide a human-readable documentation alongside (perhaps generated from the JSON). This could be an **annotated Markdown file** or web page listing each MIDI 2.0 message and field in a clean format. The idea is to double as a reference manual. When an LLM is being used, the prompt could include segments of this documentation as needed, which might be easier for it to digest than raw JSON (depending on the prompting approach). This documentation will also attract outside contributors/users who might give feedback, further improving the fidelity.

5. **Iterate with LLM in the Loop:** Once a preliminary structured spec is ready, **test it with an LLM** for the intended purpose. For example, prompt an LLM with a part of the JSON and ask it to generate Swift code for a particular message or feature. Evaluate the output – did it use the data correctly? Any misunderstandings? This will reveal if certain spec details need to be represented differently. Maybe the LLM got a field type wrong – perhaps adding a hint like `"range": "0–127"` or `"type": "uint7"` in the JSON would help. Refine the format based on these experiments to ensure it truly facilitates correct code synthesis.

6. **Continuous Update Mechanism:** Going forward, set up a mechanism to update the spec data when the MIDI Association releases new revisions or extensions. Because you've included metadata like document references and version numbers, you can quickly spot what changed in a new revision (e.g., "Spec v1.2 added two new messages and changed one enum"). The tool should be updated accordingly and perhaps note the changes. This keeps the dataset "canonical" and up-to-date. Consumers (including LLMs) will then always be referring to the latest truth.

By following this roadmap, `midi2-reader` can transform from a piecemeal parser into a **complete source of truth for MIDI 2.0**. The end result will be a resource that not only covers 100% of the spec in a machine-readable way, but does so with clarity and reliability such that anyone – human or AI – can utilize it to implement MIDI 2.0 features in Swift or any other language with confidence.

Ultimately, these improvements set the stage for LLMs to become truly effective co-pilots in MIDI 2.0 development. With a perfect specification dataset at hand, an LLM can generate Swift code that aligns exactly with official requirements, speeding up development while minimizing errors. The combination of **complete coverage, structured data, and thoughtful organization** will ensure the MIDI 2.0 spec is no longer a hundred-page PDF to decipher, but a living database that drives correct and innovative implementations [1] [2].

---

[1] [2] MIDI 2.0 Specs Released! - Art+Logic
https://artandlogic.com/midi-2-0-specs-released/

[3] [8] midi2 - Rust
https://docs.rs/midi2/latest/midi2/

4 5 6 7 Universal MIDI Packet (UMP) Format and MIDI 2.0 Protocol v1.1.1
https://amei.or.jp/midistandardcommittee/MIDI2.0/MIDI2.0-DOCS/M2-104-UM_v1-1-1_UMP_and_MIDI_2-0_Protocol_Specification.pdf