Check for updates

NIST Internal Report
NIST IR 8616

ANSI/NIST-ITL 1-2025 Machine-Readable Specification

Technical Framework and Implementation

Dylan Yaga

This publication is available free of charge from:
https://doi.org/10.6028/NIST.IR.8616

NIST NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
U.S. DEPARTMENT OF COMMERCE

NIST Internal Report
NIST IR 8616

ANSI/NIST-ITL 1-2025 Machine-Readable Specification

Technical Framework and Implementation

Dylan Yaga
Computer Security Division
Information Technology Laboratory

This publication is available free of charge from:
https://doi.org/10.6028/NIST.IR.8616

November 2026



U.S. Department of Commerce
Howard Lutnick, Secretary

National Institute of Standards and Technology
Arvind Raman, NIST Director and Under Secretary of Commerce for Standards and Technology

Certain equipment, instruments, software, or materials, commercial or non-commercial, are identified in this paper in order to specify the experimental procedure adequately. Such identification does not imply recommendation or endorsement of any product or service by NIST, nor does it imply that the materials or equipment identified are necessarily the best available for the purpose.

There may be references in this publication to other publications currently under development by NIST in accordance with its assigned statutory responsibilities. The information in this publication, including concepts and methodologies, may be used by federal agencies even before the completion of such companion publications. Thus, until each publication is completed, current requirements, guidelines, and procedures, where they exist, remain operative. For planning and transition purposes, federal agencies may wish to closely follow the development of these new publications by NIST.

Organizations are encouraged to review all draft publications during public comment periods and provide feedback to NIST. Many NIST cybersecurity publications, other than the ones noted above, are available at https://csrc.nist.gov/publications.

NIST Technical Series Policies
Copyright, Use, and Licensing Statements
NIST Technical Series Publication Identifier Syntax

Publication History
Approved by the NIST Editorial Review Board on 2026-10-23

How to Cite this NIST Technical Series Publication
Yaga D (2026) ANSI/NIST-ITL 1-2025 Machine-Readable Specification: Technical Framework and Implementation. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Internal Report (NIST IR) NIST IR 8616. https://doi.org/10.6028/NIST.IR.8616

Author ORCID iDs
Dylan Yaga: 0000-0003-4058-3645

Contact Information
ir8616-comments@nist.gov

National Institute of Standards and Technology
Attn: Computer Security Division, Information Technology Laboratory
100 Bureau Drive (Mail Stop 8930) Gaithersburg, MD 20899-8930

Additional Information
Additional information about this publication is available at https://csrc.nist.gov/pubs/ir/8616/final, including related content, potential updates, and document history.

All comments are subject to release under the Freedom of Information Act (FOIA).

Abstract

This report presents the ANSI/NIST-ITL 1-2025 requirements for exchanging biometric and forensic data using a machine-readable specification (MRS) that allows software tools to process the requirements directly. NIST developed a modular JSON architecture and an automated build process to manage, validate, and transform the MRS data using a proof-of-concept HTML specification directly from the JSON source. It demonstrates how the MRS can augment the base standard for developers, testers, and auditors to lower adoption barriers, reduce interpretation gaps, and provide a maintained digital companion to the ANSI/NIST-ITL 1-2025 specification.

Keywords

ANSI/NIST-ITL 1-2025; automated build process; biometric data interchange; data interoperability framework; deterministic implementation; JSON schema engineering; machine-readable representation; machine-readable specification (MRS); modular data architecture.

Table of Contents

1. Introduction 1
1.1. Background 1
1.2. Problem Statement 1
1.3. Purpose and Scope 2
1.4. Significance and Global Impact 2
2. Representing Requirements as Structured Data 4
3. Structured Data Development and Software Build Process 8
4. Generated HTML Specification and Data Verification 10
5. Maintenance and Collaboration 12
6. Conclusion 13
References 14
Appendix A. JSON Schema Examples 15
A.1. Example JSON Record, Field, and Related Definitions 15
A.2. JSON Abstract Syntax Tree Logic Block and Related Definitions 22
Appendix B. JSON Data Example 38

List of Tables

Table 1. Structure hierarchy 4

List of Figures

Fig. 1. Example field structure from ANSI/NIST-ITL 1-2025 5
Fig. 2. JSON constraints for field 8.006 7
Fig. 3. Automated batch script output 8
Fig. 4. Resulting generated HTML output 9
Fig. 5. HTML representation of the JSON field data 11
Fig. 6. JSON schema example for MRS data elements 22
Fig. 7. JSON schema for abstract syntax tree logic 37
Fig. 8. Example JSON data for field 8.006 40

1. Introduction

1.1. Background

The first version of the ANSI/NBS-ICST 1-1986 standard was published by NIST (then the National Bureau of Standards) in 1986 as a 53-page document. That version focused exclusively on fingerprint identification and a data format for information exchange between organizations. Since then, the standard has evolved to meet the needs of national and international law enforcement and biometric organizations by incorporating new biometric modalities, encoding formats, and metadata fields.

ANSI/NIST-ITL 1-2025, *Data Format for the Interchange of Fingerprint, Facial and Other Biometric Information* [1], is the latest revision of the global standard for biometric and forensic data. It has expanded to include guidance and data format specifications for various biometric modalities, including facial, iris, plantar, and DNA data. As the number of modalities and the complexity of the information have increased over time, the specification has grown to exceed 600 pages. Traditionally, software developers and system integrators have used this standard as a text-based reference, manually extracting and interpreting technical requirements to develop conformant software.

1.2. Problem Statement

The ANSI/NIST-ITL 1 standards have historically required extensive cross-referencing within the document. Requirements for a single data element (e.g., data type, character constraints, occurrence limits) are often distributed and repeated across narrative sections and associated tables. This fragmentation has increased the risk of ambiguity when separate sections present inconsistent or conflicting guidance, particularly in a long and complex technical specification maintained over multiple revisions. Organizations have therefore had to reconcile related requirements before implementing them, and independent reconciliation efforts could lead to divergent interpretations and subtle interoperability challenges.

NIST developed the Biometric Conformance Test Software (BioCTS) to validate whether implementations conform to the standard. BioCTS is valuable, but it is primarily a reactive tool — it identifies nonconformance after implementation work has begun. It does not negate the need to interpret requirements before design and development, nor does it prevent misunderstandings from being built into an implementation.

Before this NIST-led effort, an external initiative created “Machine-Readable Tables” for the 2011 and 2013 revisions. Those spreadsheet-based tables used proprietary software to generate XML representations of the standard’s requirements. The resources were not maintained and had limited industry adoption, so most organizations have continued to extract requirements independently from the narrative text.

1.3. Purpose and Scope

This report describes NIST's development of a machine-readable specification (MRS) for ANSI/NIST-ITL 1-2025. The MRS represents requirements from the text standard as structured JSON data. JSON was selected because it can represent the standard's hierarchy (e.g., transactions, records, fields, subfields, information items) in a form that software can validate, transform, and use directly. The MRS is intended to supplement the human-readable standard.

The scope of this report covers the process used for extracting requirements for transactions, records, fields, subfields, and information items and codifying them into a structured JSON format. This document also provides a technical overview of the supporting infrastructure, including the strongly typed JSON schema and the suite of Python scripts developed to aggregate, validate, and transform the MRS data. Furthermore, the report describes the workflow batch script used to ensure consistent, repeatable validation and the generation of a proof-of-concept HTML output page. This generated output demonstrates that the machine-readable data can accurately recreate human-readable requirements while remaining accessible to automated tools. Finally, this report outlines the strategy for releasing the project as an open-source resource. It describes how making the entirety of the effort publicly available is intended to foster a collaborative environment in which the biometric community can provide feedback, submit fixes, and contribute to the maintenance of the MRS.

The remainder of this report proceeds from data modeling to implementation:

- Section 2 describes how requirements are represented in JSON.
- Section 3 describes the build and validation process.
- Section 4 explains how generated HTML is used to review the JSON representation.
- Section 5 describes maintenance and collaboration.

1.4. Significance and Global Impact

Organizations worldwide rely on the ANSI/NIST-ITL 1 standard, including the International Criminal Police Organization (INTERPOL), the Federal Bureau of Investigation (FBI), and the Department of Homeland Security (DHS). The 2025 revision reduced the cross-reference burden for readers by placing related data specifications closer to the narrative requirements that describe them rather than distributing those details across many sections. It also resolved inconsistencies between the requirements represented in different sections of the text (e.g., within text clauses, tables, and figures).

NIST developed a JSON-encoded version of ANSI/NIST-ITL 1 to provide the global community with a single shared interpretation of the standard's complex requirements. The MRS facilitates rapid, consistent, and accurate deployments of the standard across varied technical environments and supports the exchange and integrity of biometric and forensic transactions. It can be used directly by software and adapted into implementation profiles to help minimize interoperability issues between organizations, ensure that disparate systems remain consistent and reduce the resource-intensive nature of manual requirement extraction. This will also

enable NIST and the biometric community to manage the complexity of biometric data format requirements with greater agility, including simplified auditing and the integration of community contributions (e.g., corrections, enhancements). By providing a consistent digital reference, this machine-readable representation can be reliably deployed across multiple organizations to support the implementation of the standard.

2. Representing Requirements as Structured Data

NIST developed the MRS iteratively by extracting requirements from the ANSI/NIST-ITL 1-2025 text and subsequently expanding the schema to represent new requirement patterns. When a schema feature simplified an existing representation, previously encoded JSON data was revised to use the improved structure. This approach kept the final MRS consistent across the specification.

The JSON schema establishes a hierarchy that mirrors the organization of the standard, converting text-based requirements into reusable, tokenized data structures. The schema makes use of reusable components to define data (a subset can be found in Appendix A.1 in Fig. 6 and Fig. 7) and provides formal definitions for each data level to maintain structural consistency with the base standard, as seen in Table 1.

Table 1. Structure hierarchy

MRS Level	Contains	Role
Records	Record metadata and fields	Represent top-level record types in the standard
Fields	Field metadata and subfields	Represent numbered fields within a record
Subfields	Subfield metadata and information items	Represent structured components within a field
Information Items	The most granular data elements	Represent the lowest-level requirements encoded in the MRS

These levels share common data structures so that the same validation logic can be applied consistently across the MRS. This modularity extends to data validation through a shared pattern library that houses regular expressions in a single location. By referencing these patterns throughout the MRS, the schema ensures that validation logic is applied consistently without repeating complex patterns for every instance.

The common structural elements mapped within the MRS are illustrated in Fig. 1:

- **Identifier** — Typically a record type number, a combination of a record type number and field number, or another unique identifier
- **Mnemonic** — A shorthand character string (typically 3-7 characters) used to identify fields and information items
- **Field name** — The full, formal title of the data element, as defined in the standard
- **Description** — The full textual description of the data, retained to serve as a human-readable reference within the machine-readable specification
- **Condition** — Specifies whether the data is mandatory, optional, or user-defined, often incorporating conditional logic and encoding specific rules
- **Occurrence** — Defines the number of times a data element may appear, ranging from fixed counts to dynamic values governed by conditional logic

- **Value Constraints** — Specifies allowed values, ranges, or references to external tables, often using complex conditional logic to express requirements

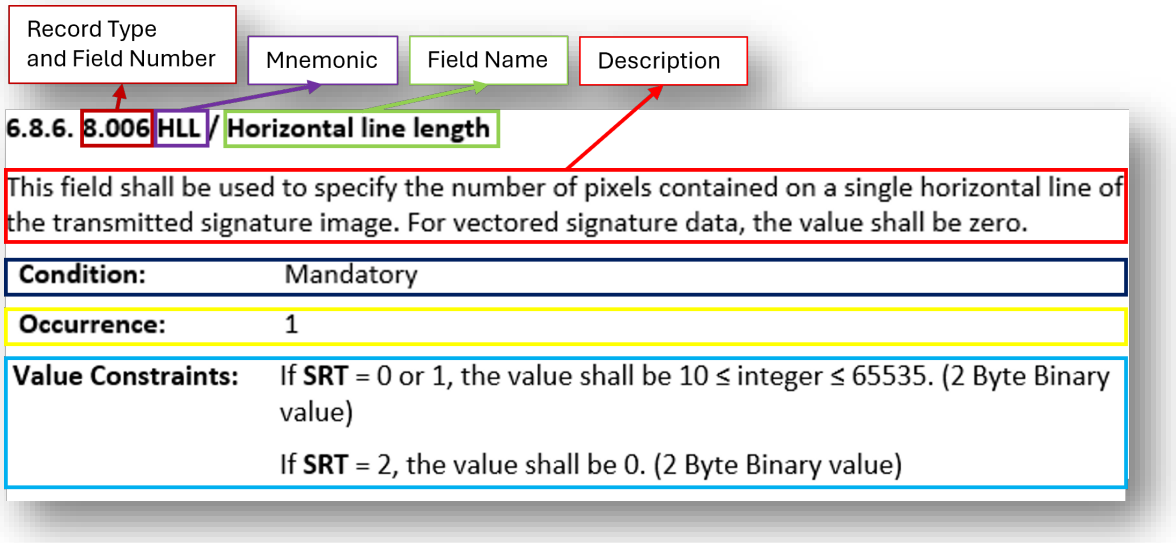


Fig. 1. Example field structure from ANSI/NIST-ITL 1-2025

Beyond structural mirroring, the JSON schema incorporates a logic-handling system to manage the standard's complex requirements. It uses specialized definitions for condition and occurrence types that allow for encoding specific rules. For example, a field may be mandatory in a traditional encoding but optional in a National Information Exchange Model (NIEM) (i.e., XML-based) encoding. These logic structures (including predicates, outcomes, and Boolean operations; see Appendix A.2) are integrated directly into the data elements. This enables the representation of conditional dependencies, where the requirements for a specific field are contingent upon the value of a separate data element. For example, the Value Constraints for field 8.006 (seen at the bottom of Fig. 1) are governed by the value of the Scan Resolution Table (SRT) field. In this example, SRT values 0 and 1 require field 8.006 to contain an integer from 10 to 65535, while SRT value 2 requires the field value to be 0. The following JSON fragment, summarized in Fig. 2, provides an example of how these conditional value constraints are formalized. Appendix B Fig. 8 provides the complete JSON data for field 8.006.

```
"constraints": {
  "conditionalValue": {
    "kind": "conditional",
    "resultType": "valueConstraint",
    "evaluation": "firstMatch",
    "onNoMatch": "error",
    "cases": [
      {
        "when": {
          "kind": "comparison",
          "op": "eq",
          "left": {
            "kind": "ref",
            "scope": "sameRecord",
            "recordType": 8,
```

```
            "fieldId": "8.004",
            "occurrence": "current",
            "value": "logical",
            "expectedType": "integer",
            "cardinality": "one",
            "mnemonic": "SRT",
            "description": "Signature representation type from Field 8.004:SRT in the same
Type-8 record."
          },
          "returnType": "boolean",
          "missingBehavior": "false",
          "collectionSemantics": "scalar",
          "typeCoercion": "none",
          "right": {
            "kind": "literal",
            "value": 2,
            "valueType": "integer",
            "cardinality": "one"
          }
        },
        "result": {
          "fixed": 0,
          "description": "If SRT = 2 (vector data), HLL shall be 0."
        }
      },
      {
        "when": {
          "kind": "logical",
          "op": "or",
          "args": [
            {
              "kind": "comparison",
              "op": "eq",
              "left": {
                "kind": "ref",
                "scope": "sameRecord",
                "recordType": 8,
                "fieldId": "8.004",
                "occurrence": "current",
                "value": "logical",
                "expectedType": "integer",
                "cardinality": "one",
                "mnemonic": "SRT",
                "description": "Signature representation type from Field 8.004:SRT in the same
Type-8 record."
              },
              "returnType": "boolean",
              "missingBehavior": "false",
              "collectionSemantics": "scalar",
              "typeCoercion": "none",
              "right": {
                "kind": "literal",
                "value": 0,
                "valueType": "integer",
                "cardinality": "one"
              }
            },
            {
              "kind": "comparison",
              "op": "eq",
              "left": {
                "kind": "ref",
                "scope": "sameRecord",
                "recordType": 8,
                "fieldId": "8.004",
                "occurrence": "current",
                "value": "logical",
                "expectedType": "integer",
                "cardinality": "one",
                "mnemonic": "SRT",
```

```
                "description": "Signature representation type from Field 8.004:SRT in the same
Type-8 record."
              },
              "returnType": "boolean",
              "missingBehavior": "false",
              "collectionSemantics": "scalar",
              "typeCoercion": "none",
              "right": {
                "kind": "literal",
                "value": 1,
                "valueType": "integer",
                "cardinality": "one"
              }
            }
          ],
          "returnType": "boolean",
          "shortCircuit": true
        },
        "result": {
          "min": 10,
          "max": 65535,
          "description": "If SRT = 0 or 1 (scanned image data), HLL shall be an integer from 10
through 65535.",
          "unit": "pixels",
          "scale": 0,
          "multipleOf": 1,
          "rounding": "exact"
        }
      }
    ],
    "description": "Logical horizontal line length constraint depends on the signature
representation type in 8.004:SRT."
  }
}
```

Fig. 2. JSON constraints for field 8.006

By breaking down text-based requirements into these discrete, formal components, the JSON specification provides a machine-verifiable mechanism for enforcing complex data constraints that are found throughout the ANSI/NIST-ITL 1-2025 standard.

3. Structured Data Development and Software Build Process

NIST developed a strongly typed JSON schema and a suite of Python scripts to aggregate, validate, and transform the data in the MRS. Python was selected for this effort because of its widespread availability across numerous platforms as well as its extensive and mature libraries for JSON processing and data validation. This ensures that the tools remain accessible to the biometric community and can effectively manage the structural complexities of the MRS.

The MRS is maintained as modular partial JSON files rather than as one monolithic document. The complete MRS is nearly 120,000 lines, and a single file proved difficult to edit in common development environments. The partial files are development modules; the build process assembles them into a complete JSON document for validation and publication. This modular structure also supports implementation profiles that include only records, fields, or other data relevant to a specific deployment.

At a high level, the batch script performs four steps: it merges the partial JSON files, validates the combined file against the schema, transforms the validated data into HTML, and post-processes the generated HTML for review. Fig. 3 shows output from this automated workflow.

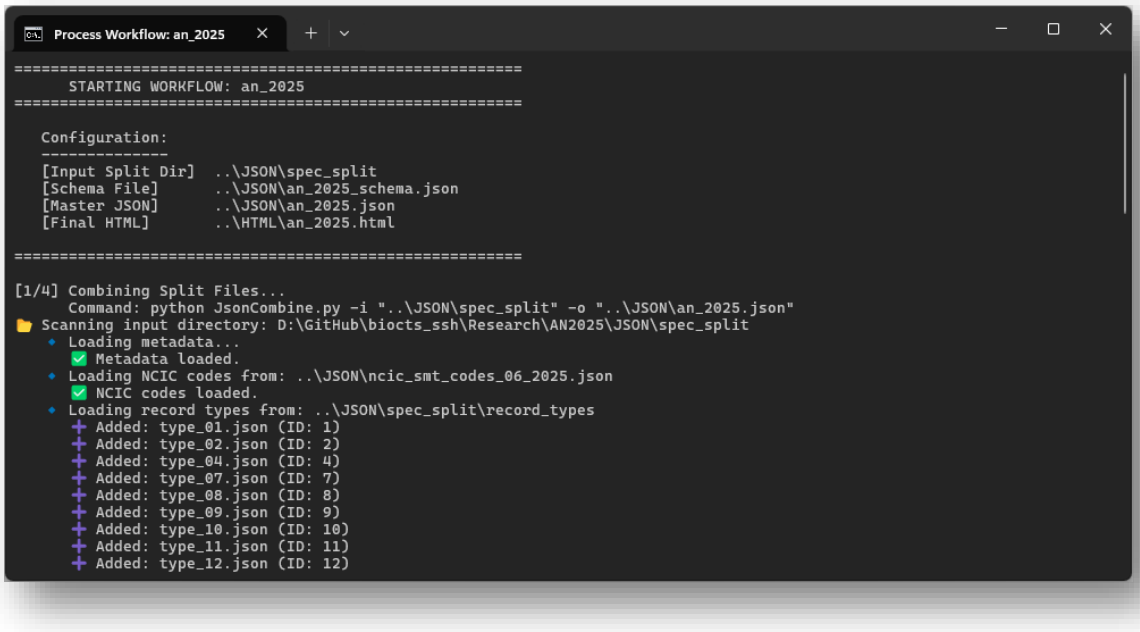


Fig. 3. Automated batch script output

The validation step checks that the combined JSON is syntactically valid and conforms to the schema. If an error occurs during validation, the Python script identifies the most relevant error match and generates a human-readable “friendly path” to the violation. This logic replaces the generic built-in data element and array indexes with descriptive labels pulled directly from the JSON data (e.g., a specific record type, field name) to help developers pinpoint and resolve mismatches more efficiently.

After successful validation, a transformation script generates an HTML representation of the standard directly from the JSON source, as illustrated in Fig. 4.



Fig. 4. Resulting generated HTML output

The script identifies references to record types, field numbers, information items, and tables and automatically applies hyperlinks to the output. This tagging improves navigation and demonstrates how the structured data can support additional forms of document automation. A final post-processing utility cleans up the HTML markup to produce an auditable, human-readable technical document from the validated JSON data.

4. Generated HTML Specification and Data Verification

The automated generation of the HTML specification serves as the primary proof-of-concept for the MRS by testing whether the data model can support reconstruction of the standard's requirements. To ensure a rigorous test, the transformation script is isolated from external narrative sources and relies exclusively on the JSON-encoded data to prevent the script from filling gaps with outside text. The resulting HTML document is a direct product of the information stored within the JSON. The successful reconstruction of complex biometric record types provides evidence that the JSON schema and data structures capture the specifications of the ANSI/NIST-ITL 1-2025 standard without omitting critical details.

Fig. 5 shows the HTML representation for field 8.006, which can be compared to the original ANSI/NIST-ITL 1-2025 text-based specification in Fig. 1.

This comparison demonstrates that the requirements for field 8.006 have been captured in the JSON and reproduced in the generated HTML. Similar comparisons across record types can be used to review coverage across the complete MRS.

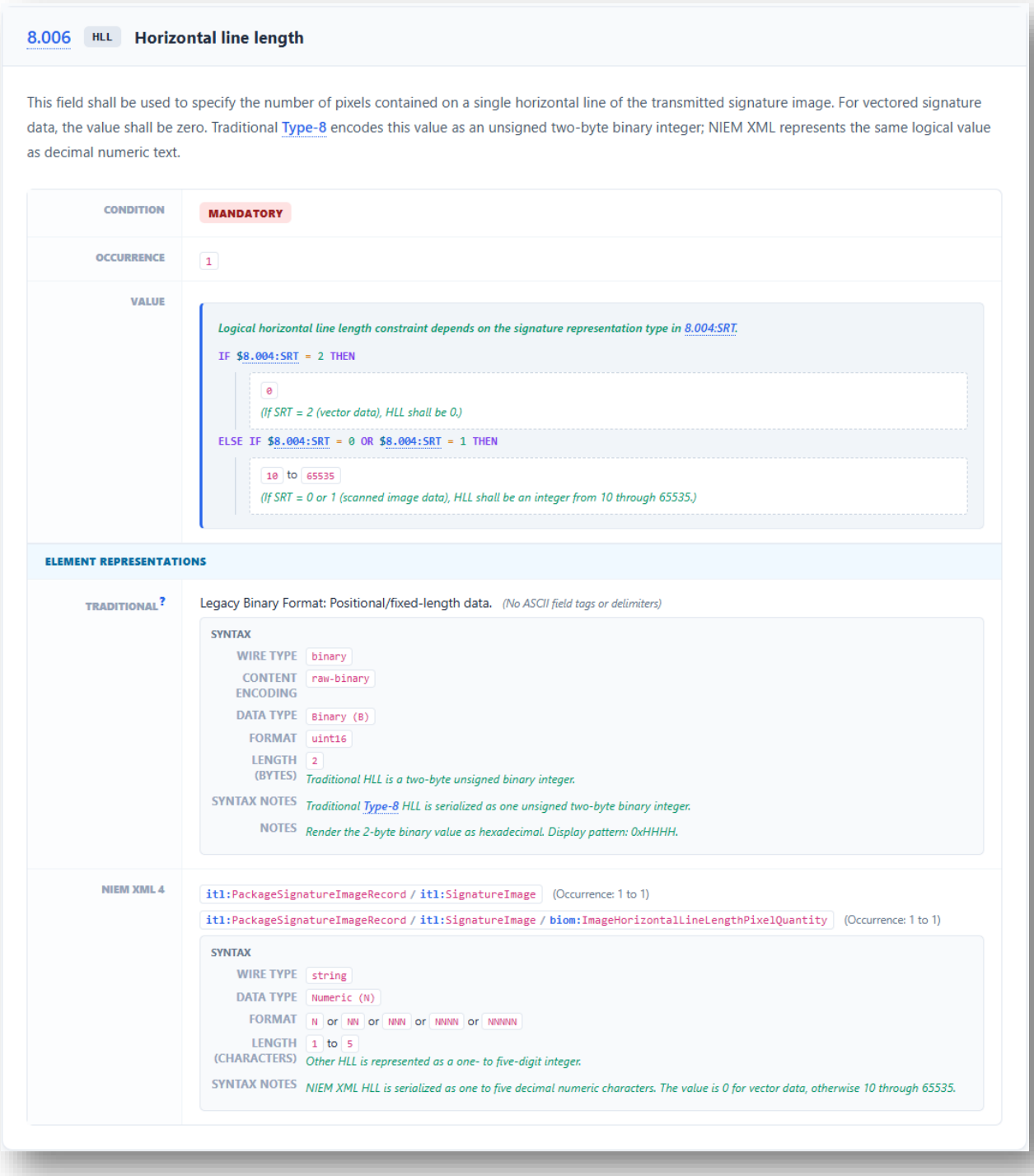


Fig. 5. HTML representation of the JSON field data

5. Maintenance and Collaboration

To encourage adoption and keep the MRS current, NIST is releasing the project as an open-source repository. This public repository will allow users to propose corrections, report issues, and suggest enhancements. Users may also copy the repository for internal use. Changes to the NIST-maintained version, however, must pass schema validation and subject-matter review before incorporation.

The automated validation scripts serve as an initial technical filter to ensure that any proposed update is structured correctly, follows the schema, and can complete the entire build process. A successful build only confirms that the data is well-formed; it does not mean a change will be accepted. While expert review and technical consensus remain necessary, using the JSON data and the automated build process allows any tool or transformation that uses the MRS to reflect updates as soon as they are approved and incorporated.

6. Conclusion

The MRS for ANSI/NIST-ITL 1-2025 translates text-based biometric data exchange requirements into structured JSON. By converting more than 600 pages of requirements into machine-processable data, NIST has reduced the need for manual extraction and lowered the risk of inconsistent interpretation. This project uses modular JSON files, a JSON schema, and automated build scripts to assemble and validate the MRS. NIST expert review remains necessary to ensure that the JSON accurately represents the base standard.

Together, the schema, validation scripts, generated HTML output, and expert review process establish a repeatable workflow for maintaining the MRS as a technical resource and community-supported reference. The resulting JSON data can support implementation, testing, and auditing tools while preserving NIST review of the official specification.

References

[1] Stathakis J, Fiumara G, Orandi S, Marshall K (2026), Information Technology: Data Format for the Interchange of Fingerprint, Facial and Other Biometric Information. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) NIST SP 500-290e4. https://doi.org/10.6028/NIST.SP.500-290e4

Appendix A. JSON Schema Examples

The examples in the appendices are the development snapshot at the time of drafting this document and may have been modified as the project continued.

A.1. Example JSON Record, Field, and Related Definitions

```
{
  "recordType": {
    "type": "object",
    "required": [
      "id",
      "name",
      "description",
      "fields"
    ],
    "properties": {
      "id": {
        "type": "integer"
      },
      "name": {
        "type": "string",
        "description": "The name of the record type."
      },
      "description": {
        "type": "string",
        "description": "A detailed description of the record type's purpose."
      },
      "fields": {
        "type": "array",
        "minItems": 1,
        "description": "List of fields and feature sets contained within the record type.",
        "items": {
          "anyOf": [
            {
              "$ref": "#/$defs/field"
            },
            {
              "$ref": "#/$defs/limitedDataField"
            },
            {
              "$ref": "#/$defs/featureSet"
            }
          ]
        }
      },
      "representations": {
        "$ref": "#/$defs/encodingRepresentations",
        "description": "Record-level representation bundles, typically root/container paths per
encoding."
      }
    },
    "additionalProperties": false
  },
  "dataElement": {
    "$comment": "Mixin: The base properties shared by fields and information items",
    "properties": {
      "mnemonic": {
        "type": "string"
      },
      "name": {
        "type": "string"
      },
      "description": {
        "type": "string"
      },
      "condition": {
```

```
        "$ref": "#/$defs/conditionType"
      },
      "occurrence": {
        "$ref": "#/$defs/occurrenceType"
      },
      "constraints": {
        "$ref": "#/$defs/constraints"
      },
      "figures": {
        "type": "array",
        "items": {
          "type": "object",
          "required": [
            "figure"
          ],
          "properties": {
            "figure": {
              "$ref": "#/$defs/figure"
            }
          },
          "additionalProperties": false
        }
      },
      "representations": {
        "$ref": "#/$defs/encodingRepresentations",
        "description": "Encoding-specific representation bundles for this logical data
element."
      }
    }
  },
  "field": {
    "type": "object",
    "oneOf": [
      {
        "required": [
          "id"
        ]
      },
      {
        "required": [
          "startId",
          "endId"
        ]
      }
    ],
    "required": [
      "mnemonic",
      "name",
      "description"
    ],
    "properties": {
      "id": {
        "$ref": "#/$defs/idType"
      },
      "startId": {
        "$ref": "#/$defs/idType"
      },
      "endId": {
        "$ref": "#/$defs/idType"
      },
      "mnemonic": {
        "type": "string"
      },
      "name": {
        "type": "string"
      },
      "description": {
        "type": "string"
      },
      "condition": {
        "$ref": "#/$defs/conditionType"
```

```
      },
      "occurrence": {
        "$ref": "#/$defs/occurrenceType"
      },
      "constraints": {
        "$ref": "#/$defs/constraints"
      },
      "figures": {
        "type": "array",
        "items": {
          "type": "object",
          "required": [
            "figure"
          ],
          "properties": {
            "figure": {
              "$ref": "#/$defs/figure"
            }
          },
          "additionalProperties": false
        }
      },
      "representations": {
        "$ref": "#/$defs/encodingRepresentations",
        "description": "Encoding-specific representation bundles for this logical data
element."
      },
      "subfields": {
        "type": "array",
        "items": {
          "$ref": "#/$defs/subfieldSet"
        }
      }
    },
    "additionalProperties": false
  },
  "limitedDataField": {
    "type": "object",
    "oneOf": [
      {
        "required": [
          "id"
        ]
      },
      {
        "required": [
          "startId",
          "endId"
        ]
      }
    ],
    "required": [
      "mnemonic",
      "name",
      "description"
    ],
    "properties": {
      "id": {
        "$ref": "#/$defs/idType"
      },
      "startId": {
        "$ref": "#/$defs/idType"
      },
      "endId": {
        "$ref": "#/$defs/idType"
      },
      "mnemonic": {
        "type": "string"
      },
      "name": {
        "type": "string"
```

```
      },
      "description": {
        "type": "string"
      },
      "representations": {
        "$ref": "#/$defs/encodingRepresentations",
        "description": "Encoding-specific representation bundles for this limited data field."
      }
    },
    "additionalProperties": false
  },
  "subfieldSet": {
    "type": "object",
    "required": [
      "occurrence",
      "structure",
      "informationItems"
    ],
    "properties": {
      "name": {
        "type": "string"
      },
      "description": {
        "type": "string"
      },
      "condition": {
        "$ref": "#/$defs/conditionType"
      },
      "occurrence": {
        "$ref": "#/$defs/occurrenceType",
        "description": "Number of times this subfield set/template occurs within the containing
field when that field is present."
      },
      "structure": {
        "$ref": "#/$defs/subfieldStructure",
        "description": "Identifies whether this subfield set is fixed-position static content
or a repeating subfield template."
      },
      "position": {
        "$ref": "#/$defs/zeroBasedPosition",
        "description": "Optional zero-based position metadata for this subfield set/template."
      },
      "informationItems": {
        "type": "array",
        "minItems": 1,
        "items": {
          "$ref": "#/$defs/informationItemEntry"
        },
        "description": "Ordered traditional information-item layout. Entries may be single
information items or nested information-item sets."
      },
      "representations": {
        "$ref": "#/$defs/encodingRepresentations",
        "description": "Encoding-specific representation bundles for this subfield group."
      },
      "presenceAssertions": {
        "type": "array",
        "items": {
          "$ref": "#/$defs/subfieldPresenceAssertion"
        },
        "description": "Machine-readable subfield-level information-item presence assertions
that cannot be expressed by individual item occurrence rules."
      }
    },
    "additionalProperties": false
  },
  "subfieldStructure": {
    "type": "object",
    "required": [
      "kind"
    ],
```

```
    "properties": {
      "kind": {
        "type": "string",
        "enum": [
          "static",
          "repeating"
        ],
        "description": "Static subfields occupy a fixed position in the subfield array;
repeating subfields describe a repeatable subfield template."
      }
    },
    "additionalProperties": false,
    "description": "Machine-readable subfield classification."
  },
  "subfieldPresenceAssertion": {
    "type": "object",
    "required": [
      "kind",
      "minPresent"
    ],
    "properties": {
      "kind": {
        "const": "informationItemPresence",
        "description": "Counts populated information items selected from the current subfield."
      },
      "name": {
        "type": "string"
      },
      "description": {
        "type": "string"
      },
      "encodings": {
        "type": "array",
        "items": {
          "$ref": "#/$defs/encodingId"
        },
        "description": "Optional encoding IDs to which this assertion applies. Omit for all
encodings."
      },
      "minPresent": {
        "type": "integer",
        "minimum": 0
      },
      "maxPresent": {
        "type": "integer",
        "minimum": 0
      },
      "mnemonics": {
        "type": "array",
        "items": {
          "type": "string"
        },
        "description": "Explicit information-item mnemonics to count. If omitted, all items in
the selected range are counted."
      },
      "excludeMnemonics": {
        "type": "array",
        "items": {
          "type": "string"
        },
        "description": "Information-item mnemonics to exclude from the count."
      },
      "afterMnemonic": {
        "type": "string",
        "description": "Count only information items positioned after the first item with this
mnemonic."
      },
      "errorCode": {
        "type": "string"
      },
      "passCode": {
```

```
        "type": "string"
      },
      "failureMessage": {
        "type": "string"
      },
      "passMessage": {
        "type": "string"
      }
    },
    "additionalProperties": false,
    "description": "Subfield-level presence rule for assertions such as “at least one
information item other than SIL shall be present.”"
  },
  "informationItem": {
    "type": "object",
    "required": [
      "name",
      "description"
    ],
    "properties": {
      "mnemonic": {
        "type": "string"
      },
      "name": {
        "type": "string"
      },
      "description": {
        "type": "string"
      },
      "condition": {
        "$ref": "#/$defs/conditionType"
      },
      "occurrence": {
        "$ref": "#/$defs/occurrenceType"
      },
      "constraints": {
        "$ref": "#/$defs/constraints"
      },
      "figures": {
        "type": "array",
        "items": {
          "type": "object",
          "required": [
            "figure"
          ],
          "properties": {
            "figure": {
              "$ref": "#/$defs/figure"
            }
          },
          "additionalProperties": false
        }
      },
      "representations": {
        "$ref": "#/$defs/encodingRepresentations",
        "description": "Encoding-specific representation bundles for this logical data
element."
      }
    },
    "additionalProperties": false
  },
  "informationItemEntry": {
    "$comment": "An entry in subfieldSet.informationItems can be either one information item or
a nested information-item set.",
    "oneOf": [
      {
        "$ref": "#/$defs/informationItem"
      },
      {
        "$ref": "#/$defs/informationItemSet"
      }
```

```
      ]
    },
    "informationItemSet": {
      "type": "object",
      "required": [
        "type",
        "name",
        "description",
        "informationItems"
      ],
      "properties": {
        "type": {
          "const": "informationItemSet",
          "description": "Discriminator identifying this entry as a nested set of information
items rather than a single information item."
        },
        "id": {
          "type": "string",
          "pattern": "^[A-Za-z_][A-Za-z0-9_.-]*$",
          "description": "Stable identifier for this information-item set within the subfield."
        },
        "name": {
          "type": "string"
        },
        "description": {
          "type": "string"
        },
        "condition": {
          "$ref": "#/$defs/conditionType",
          "description": "Condition for the set as a whole. If omitted, the set is applicable
whenever the containing subfield is applicable."
        },
        "occurrence": {
          "$ref": "#/$defs/occurrenceType",
          "description": "Number of times this set occurs. For repeating traditional groups, use
dynamicFixed with a structured reference expression or a min/max range."
        },
        "notes": {
          "type": "string"
        },
        "informationItems": {
          "type": "array",
          "minItems": 1,
          "items": {
            "$ref": "#/$defs/informationItem"
          }
        },
        "representations": {
          "$ref": "#/$defs/encodingRepresentations",
          "description": "Encoding-specific representation bundles for this information-item set,
including layout/separator rules."
        }
      },
      "additionalProperties": false
    },
    "featureSet": {
      "type": "object",
      "required": [
        "name",
        "fields"
      ],
      "properties": {
        "name": {
          "type": "string",
          "description": "Name of the feature set (e.g., 'INCITS_378', 'EFS')"
        },
        "description": {
          "type": "string",
          "description": "Description of the feature set and its conditional requirements"
        },
        "fields": {
```

```
        "type": "array",
        "minItems": 1,
        "items": {
          "anyOf": [
            {
              "$ref": "#/$defs/field"
            },
            {
              "$ref": "#/$defs/limitedDataField"
            }
          ]
        }
      },
      "representations": {
        "$ref": "#/$defs/encodingRepresentations",
        "description": "Optional feature-set-level representation bundles."
      }
    },
    "additionalProperties": false
  },
  "identifiable": {
    "$comment": "Mixin: Requires either an 'id' or a 'startId'/'endId' pair",
    "properties": {
      "id": {
        "$ref": "#/$defs/idType"
      },
      "startId": {
        "$ref": "#/$defs/idType"
      },
      "endId": {
        "$ref": "#/$defs/idType"
      }
    },
    "oneOf": [
      {
        "required": [
          "id"
        ]
      },
      {
        "required": [
          "startId",
          "endId"
        ]
      }
    ]
  }
}
```

Fig. 6. JSON schema example for MRS data elements

A.2. JSON Abstract Syntax Tree Logic Block and Related Definitions

```
{
  "expression": {
    "$comment": "All executable expression nodes are kind-discriminated. Primitive literals are
represented by literalExpression nodes and references by referenceExpression nodes.",
    "oneOf": [
      {
        "$ref": "#/$defs/literalExpression"
      },
      {
        "$ref": "#/$defs/referenceExpression"
      },
      {
        "$ref": "#/$defs/callExpression"
      },
      {
        "$ref": "#/$defs/comparisonExpression"
```

```
      },
      {
        "$ref": "#/$defs/logicalExpression"
      },
      {
        "$ref": "#/$defs/mathExpression"
      }
    ]
  },
  "predicate": {
    "$ref": "#/$defs/booleanExpression"
  },
  "literalExpression": {
    "type": "object",
    "required": [
      "kind",
      "value"
    ],
    "properties": {
      "kind": {
        "const": "literal"
      },
      "value": {
        "$ref": "#/$defs/literal"
      },
      "valueType": {
        "$ref": "#/$defs/dataType",
        "description": "Optional explicit type. If omitted, the evaluator infers the type from
value."
      },
      "cardinality": {
        "$ref": "#/$defs/cardinality",
        "default": "one"
      },
      "description": {
        "type": "string"
      }
    },
    "additionalProperties": false,
    "description": "Literal expression node. Primitive literals are wrapped so every executable
expression has a kind discriminator."
  },
  "referenceExpression": {
    "type": "object",
    "required": [
      "kind",
      "scope"
    ],
    "properties": {
      "kind": {
        "const": "ref"
      },
      "scope": {
        "type": "string",
        "enum": [
          "currentItem",
          "currentOccurrence",
          "sameSubfield",
          "sameField",
          "sameRecord",
          "record",
          "transaction",
          "encoding",
          "profile",
          "global",
          "context"
        ],
        "description": "Evaluation scope used to resolve the reference."
      },
      "recordType": {
        "oneOf": [
```

```
            {
              "type": "integer"
            },
            {
              "type": "string"
            }
          ],
          "description": "Record type identifier when resolution is tied to a specific record
type."
        },
        "fieldId": {
          "$ref": "#/$defs/idType"
        },
        "mnemonic": {
          "type": "string",
          "description": "Field or information-item mnemonic."
        },
        "name": {
          "type": "string",
          "description": "Named context variable, such as RECORD_TYPES."
        },
        "path": {
          "type": "string",
          "description": "Encoding-neutral or evaluator-defined structured path."
        },
        "occurrence": {
          "type": "string",
          "enum": [
            "current",
            "any",
            "all",
            "first",
            "last",
            "indexed",
            "matching"
          ],
          "default": "current",
          "description": "Which occurrence(s) are selected when multiple values exist."
        },
        "index": {
          "oneOf": [
            {
              "type": "integer",
              "minimum": 0
            },
            {
              "$ref": "#/$defs/numericExpression"
            }
          ],
          "description": "Zero-based occurrence index when occurrence is indexed."
        },
        "value": {
          "type": "string",
          "enum": [
            "logical",
            "encoded",
            "wire",
            "computed",
            "presence",
            "count"
          ],
          "default": "logical",
          "description": "Which value view is requested."
        },
        "encodingId": {
          "$ref": "#/$defs/encodingId",
          "description": "Required when value is encoded or wire and the encoding is not
otherwise implied."
        },
        "expectedType": {
          "$ref": "#/$defs/dataType",
```

```
        "description": "Expected type after reference resolution."
      },
      "cardinality": {
        "$ref": "#/$defs/cardinality",
        "description": "Expected cardinality after reference resolution."
      },
      "defaultWhenMissing": {
        "$ref": "#/$defs/literal"
      },
      "description": {
        "type": "string"
      }
    },
    "additionalProperties": false,
    "description": "Structured evaluator reference with explicit scope, occurrence, and value-
view semantics.",
    "allOf": [
      {
        "anyOf": [
          {
            "required": [
              "name"
            ]
          },
          {
            "required": [
              "fieldId"
            ]
          },
          {
            "required": [
              "mnemonic"
            ]
          },
          {
            "required": [
              "path"
            ]
          }
        ]
      },
      {
        "if": {
          "properties": {
            "occurrence": {
              "const": "indexed"
            }
          },
          "required": [
            "occurrence"
          ]
        },
        "then": {
          "required": [
            "index"
          ]
        }
      }
    ]
  },
  "callExpression": {
    "type": "object",
    "required": [
      "kind",
      "name",
      "args",
      "returnType"
    ],
    "properties": {
      "kind": {
        "const": "call"
```

```
      },
      "name": {
        "$ref": "#/$defs/functionName"
      },
      "args": {
        "type": "array",
        "items": {
          "$ref": "#/$defs/expression"
        }
      },
      "returnType": {
        "$ref": "#/$defs/dataType"
      },
      "returnCardinality": {
        "$ref": "#/$defs/cardinality",
        "default": "one"
      },
      "onMissing": {
        "$ref": "#/$defs/missingBehavior",
        "default": "error"
      },
      "deterministic": {
        "type": "boolean",
        "const": true,
        "description": "Evaluator functions used by the standard must be deterministic and
side-effect-free."
      },
      "description": {
        "type": "string"
      }
    },
    "additionalProperties": false,
    "description": "Function call expression. Function name, argument count/type, return type,
and missing-value behavior must agree with evaluatorCatalog.",
    "allOf": [
      {
        "if": {
          "properties": {
            "name": {
              "const": "presence.present"
            }
          },
          "required": [
            "name"
          ]
        },
        "then": {
          "properties": {
            "args": {
              "minItems": 1
            },
            "returnType": {
              "const": "boolean"
            }
          }
        }
      },
      {
        "if": {
          "properties": {
            "name": {
              "const": "presence.notPresent"
            }
          },
          "required": [
            "name"
          ]
        },
        "then": {
          "properties": {
            "args": {
```

```
            "minItems": 1
          },
          "returnType": {
            "const": "boolean"
          }
        }
      }
    },
    {
      "if": {
        "properties": {
          "name": {
            "const": "date.isPartialDate"
          }
        },
        "required": [
          "name"
        ]
      },
      "then": {
        "properties": {
          "args": {
            "minItems": 1,
            "maxItems": 1
          },
          "returnType": {
            "const": "boolean"
          }
        }
      }
    },
    {
      "if": {
        "properties": {
          "name": {
            "const": "media.isImage"
          }
        },
        "required": [
          "name"
        ]
      },
      "then": {
        "properties": {
          "args": {
            "minItems": 1,
            "maxItems": 1
          },
          "returnType": {
            "const": "boolean"
          }
        }
      }
    },
    {
      "if": {
        "properties": {
          "name": {
            "const": "media.isVideo"
          }
        },
        "required": [
          "name"
        ]
      },
      "then": {
        "properties": {
          "args": {
            "minItems": 1,
            "maxItems": 1
          },
```

```
        "returnType": {
          "const": "boolean"
        }
      }
    }
  },
  {
    "if": {
      "properties": {
        "name": {
          "const": "media.isAudio"
        }
      },
      "required": [
        "name"
      ]
    },
    "then": {
      "properties": {
        "args": {
          "minItems": 1,
          "maxItems": 1
        },
        "returnType": {
          "const": "boolean"
        }
      }
    }
  },
  {
    "if": {
      "properties": {
        "name": {
          "const": "image.isStitchedImage"
        }
      },
      "required": [
        "name"
      ]
    },
    "then": {
      "properties": {
        "args": {
          "minItems": 2,
          "maxItems": 2
        },
        "returnType": {
          "const": "boolean"
        }
      }
    }
  },
  {
    "if": {
      "properties": {
        "name": {
          "const": "image.hasMatchingType13Image"
        }
      },
      "required": [
        "name"
      ]
    },
    "then": {
      "properties": {
        "args": {
          "minItems": 0,
          "maxItems": 0
        },
        "returnType": {
          "const": "boolean"
```

```
            }
          }
        }
      },
      {
        "if": {
          "properties": {
            "name": {
              "const": "image.matchingType13ImageIsKnownLateralReversal"
            }
          },
          "required": [
            "name"
          ]
        },
        "then": {
          "properties": {
            "args": {
              "minItems": 0,
              "maxItems": 0
            },
            "returnType": {
              "const": "boolean"
            }
          }
        }
      },
      {
        "if": {
          "properties": {
            "name": {
              "const": "comparison.isComplex"
            }
          },
          "required": [
            "name"
          ]
        },
        "then": {
          "properties": {
            "args": {
              "minItems": 0,
              "maxItems": 0
            },
            "returnType": {
              "const": "boolean"
            }
          }
        }
      },
      {
        "if": {
          "properties": {
            "name": {
              "const": "conversion.isAnalogToDigital"
            }
          },
          "required": [
            "name"
          ]
        },
        "then": {
          "properties": {
            "args": {
              "minItems": 1,
              "maxItems": 1
            },
            "returnType": {
              "const": "boolean"
            }
          }
```

```
        }
      }
    ]
  },
  "comparisonExpression": {
    "type": "object",
    "required": [
      "kind",
      "op",
      "left",
      "returnType"
    ],
    "properties": {
      "kind": {
        "const": "comparison"
      },
      "op": {
        "type": "string",
        "enum": [
          "eq",
          "ne",
          "gt",
          "lt",
          "gte",
          "lte",
          "contains",
          "in",
          "intersects",
          "containsAny",
          "matches",
          "matchesAny",
          "isSubsetOf",
          "exists",
          "isEmpty"
        ],
        "description": "Typed comparison operator. Use eq/ne instead of ==/!= to avoid
language-specific ambiguity."
      },
      "left": {
        "$ref": "#/$defs/expression"
      },
      "right": {
        "$ref": "#/$defs/expression"
      },
      "returnType": {
        "const": "boolean"
      },
      "typeCoercion": {
        "type": "string",
        "enum": [
          "none",
          "numeric",
          "string",
          "boolean",
          "standardNumericString"
        ],
        "default": "none",
        "description": "Explicitly states whether operands may be coerced before comparison.
Default is strict no coercion."
      },
      "missingBehavior": {
        "$ref": "#/$defs/missingBehavior",
        "default": "false"
      },
      "collectionSemantics": {
        "type": "string",
        "enum": [
          "scalar",
          "leftCollection",
          "rightCollection",
          "bothCollections"
```

```
        ],
        "description": "Documents operand cardinality expectations for collection operators."
      },
      "description": {
        "type": "string"
      }
    },
    "additionalProperties": false,
    "description": "Boolean comparison node with fixed operand order and explicit
coercion/missing-value semantics.",
    "allOf": [
      {
        "if": {
          "properties": {
            "op": {
              "enum": [
                "exists",
                "isEmpty"
              ]
            }
          },
          "required": [
            "op"
          ]
        },
        "then": {
          "not": {
            "required": [
              "right"
            ]
          }
        }
      },
      {
        "if": {
          "properties": {
            "op": {
              "not": {
                "enum": [
                  "exists",
                  "isEmpty"
                ]
              }
            }
          },
          "required": [
            "op"
          ]
        },
        "then": {
          "required": [
            "right"
          ]
        }
      }
    ]
  },
  "logicalExpression": {
    "type": "object",
    "required": [
      "kind",
      "op",
      "args",
      "returnType"
    ],
    "properties": {
      "kind": {
        "const": "logical"
      },
      "op": {
        "type": "string",
```

```
        "enum": [
          "and",
          "or",
          "not"
        ]
      },
      "args": {
        "type": "array",
        "items": {
          "$ref": "#/$defs/booleanExpression"
        }
      },
      "returnType": {
        "const": "boolean"
      },
      "shortCircuit": {
        "type": "boolean",
        "default": true,
        "description": "Whether evaluators may short-circuit left-to-right."
      },
      "description": {
        "type": "string"
      }
    },
    "additionalProperties": false,
    "description": "Boolean logical expression with an explicit operator, typed arguments, and
evaluator return type.",
    "allOf": [
      {
        "if": {
          "properties": {
            "op": {
              "const": "not"
            }
          },
          "required": [
            "op"
          ]
        },
        "then": {
          "properties": {
            "args": {
              "minItems": 1,
              "maxItems": 1
            }
          }
        }
      },
      {
        "if": {
          "properties": {
            "op": {
              "enum": [
                "and",
                "or"
              ]
            }
          },
          "required": [
            "op"
          ]
        },
        "then": {
          "properties": {
            "args": {
              "minItems": 2
            }
          }
        }
      }
    ]
```

```
  },
  "mathExpression": {
    "type": "object",
    "required": [
      "kind",
      "op",
      "args",
      "returnType"
    ],
    "properties": {
      "kind": {
        "const": "math"
      },
      "op": {
        "type": "string",
        "enum": [
          "add",
          "subtract",
          "multiply",
          "divide",
          "modulo",
          "ceiling",
          "floor",
          "round",
          "min",
          "max",
          "abs"
        ]
      },
      "args": {
        "type": "array",
        "items": {
          "$ref": "#/$defs/numericExpression"
        }
      },
      "returnType": {
        "type": "string",
        "enum": [
          "integer",
          "decimal",
          "number"
        ]
      },
      "divisionBehavior": {
        "type": "string",
        "enum": [
          "errorOnDivideByZero",
          "returnNullOnDivideByZero"
        ],
        "default": "errorOnDivideByZero"
      },
      "roundingMode": {
        "type": "string",
        "enum": [
          "ceiling",
          "floor",
          "halfUp",
          "halfEven",
          "truncate",
          "exact"
        ],
        "description": "Required when op is round; optional documentation for other numeric
operations."
      },
      "description": {
        "type": "string"
      }
    },
    "additionalProperties": false,
    "description": "Numeric math expression with one operator name, typed arguments, and
evaluator return type.",
```

```
  "allOf": [
    {
      "if": {
        "properties": {
          "op": {
            "enum": [
              "ceiling",
              "floor",
              "abs"
            ]
          }
        },
        "required": [
          "op"
        ]
      },
      "then": {
        "properties": {
          "args": {
            "minItems": 1,
            "maxItems": 1
          }
        }
      }
    },
    {
      "if": {
        "properties": {
          "op": {
            "const": "round"
          }
        },
        "required": [
          "op"
        ]
      },
      "then": {
        "properties": {
          "args": {
            "minItems": 1,
            "maxItems": 2
          }
        },
        "required": [
          "roundingMode"
        ]
      }
    },
    {
      "if": {
        "properties": {
          "op": {
            "enum": [
              "subtract",
              "divide",
              "modulo"
            ]
          }
        },
        "required": [
          "op"
        ]
      },
      "then": {
        "properties": {
          "args": {
            "minItems": 2,
            "maxItems": 2
          }
        }
      }
    }
```

```
      },
      {
        "if": {
          "properties": {
            "op": {
              "enum": [
                "add",
                "multiply",
                "min",
                "max"
              ]
            }
          },
          "required": [
            "op"
          ]
        },
        "then": {
          "properties": {
            "args": {
              "minItems": 2
            }
          }
        }
      }
    ]
  },
  "numericExpression": {
    "oneOf": [
      {
        "$ref": "#/$defs/mathExpression"
      },
      {
        "allOf": [
          {
            "$ref": "#/$defs/literalExpression"
          },
          {
            "properties": {
              "value": {
                "type": "number"
              },
              "valueType": {
                "enum": [
                  "integer",
                  "decimal",
                  "number"
                ]
              }
            },
            "required": [
              "value"
            ]
          }
        ]
      },
      {
        "allOf": [
          {
            "$ref": "#/$defs/referenceExpression"
          },
          {
            "properties": {
              "expectedType": {
                "enum": [
                  "integer",
                  "decimal",
                  "number"
                ]
              }
            },
```

```
            "required": [
              "expectedType"
            ]
          }
        ]
      },
      {
        "allOf": [
          {
            "$ref": "#/$defs/callExpression"
          },
          {
            "properties": {
              "returnType": {
                "enum": [
                  "integer",
                  "decimal",
                  "number"
                ]
              }
            },
            "required": [
              "returnType"
            ]
          }
        ]
      }
    ]
  },
  "booleanExpression": {
    "oneOf": [
      {
        "$ref": "#/$defs/comparisonExpression"
      },
      {
        "$ref": "#/$defs/logicalExpression"
      },
      {
        "allOf": [
          {
            "$ref": "#/$defs/literalExpression"
          },
          {
            "properties": {
              "value": {
                "type": "boolean"
              },
              "valueType": {
                "const": "boolean"
              }
            },
            "required": [
              "value"
            ]
          }
        ]
      },
      {
        "allOf": [
          {
            "$ref": "#/$defs/referenceExpression"
          },
          {
            "properties": {
              "expectedType": {
                "const": "boolean"
              }
            },
            "required": [
              "expectedType"
            ]
```

```
            }
          ]
        },
        {
          "allOf": [
            {
              "$ref": "#/$defs/callExpression"
            },
            {
              "properties": {
                "returnType": {
                  "const": "boolean"
                }
              },
              "required": [
                "returnType"
              ]
            }
          ]
        }
      ]
    }
  }
```

Fig. 7. JSON schema for abstract syntax tree logic

Appendix B. JSON Data Example

The examples in the appendices are the development snapshot at the time of drafting this document and may have been modified as the project continued. The following JSON is the field 8.006 requirements in machine-readable format.


```
{
      "id": "8.006",
      "mnemonic": "HLL",
      "name": "Horizontal line length",
      "description": "This field shall be used to specify the number of pixels contained on a
single horizontal line of the transmitted signature image. For vectored signature data, the
value shall be zero. Traditional Type-8 encodes this value as an unsigned two-byte binary
integer; NIEM XML represents the same logical value as decimal numeric text.",
      "condition": {
        "type": "Mandatory"
      },
      "occurrence": {
        "fixed": 1
      },
      "constraints": {
        "conditionalValue": {
          "kind": "conditional",
          "resultType": "valueConstraint",
          "evaluation": "firstMatch",
          "onNoMatch": "error",
          "cases": [
            {
              "when": {
                "kind": "comparison",
                "op": "eq",
                "left": {
                  "kind": "ref",
                  "scope": "sameRecord",
                  "recordType": 8,
                  "fieldId": "8.004",
                  "occurrence": "current",
                  "value": "logical",
                  "expectedType": "integer",
                  "cardinality": "one",
                  "mnemonic": "SRT",
                  "description": "Signature representation type from Field 8.004:SRT in the
same Type-8 record."
                },
                "returnType": "boolean",
                "missingBehavior": "false",
                "collectionSemantics": "scalar",
                "typeCoercion": "none",
                "right": {
                  "kind": "literal",
                  "value": 2,
                  "valueType": "integer",
                  "cardinality": "one"
                }
              },
              "result": {
                "fixed": 0,
                "description": "If SRT = 2 (vector data), HLL shall be 0."
              }
            },
            {
              "when": {
                "kind": "logical",
                "op": "or",
                "args": [
                  {
                    "kind": "comparison",
                    "op": "eq",
                    "left": {
```

```
                      "kind": "ref",
                      "scope": "sameRecord",
                      "recordType": 8,
                      "fieldId": "8.004",
                      "occurrence": "current",
                      "value": "logical",
                      "expectedType": "integer",
                      "cardinality": "one",
                      "mnemonic": "SRT",
                      "description": "Signature representation type from Field 8.004:SRT in the
same Type-8 record."
                    },
                    "returnType": "boolean",
                    "missingBehavior": "false",
                    "collectionSemantics": "scalar",
                    "typeCoercion": "none",
                    "right": {
                      "kind": "literal",
                      "value": 0,
                      "valueType": "integer",
                      "cardinality": "one"
                    }
                  },
                  {
                    "kind": "comparison",
                    "op": "eq",
                    "left": {
                      "kind": "ref",
                      "scope": "sameRecord",
                      "recordType": 8,
                      "fieldId": "8.004",
                      "occurrence": "current",
                      "value": "logical",
                      "expectedType": "integer",
                      "cardinality": "one",
                      "mnemonic": "SRT",
                      "description": "Signature representation type from Field 8.004:SRT in the
same Type-8 record."
                    },
                    "returnType": "boolean",
                    "missingBehavior": "false",
                    "collectionSemantics": "scalar",
                    "typeCoercion": "none",
                    "right": {
                      "kind": "literal",
                      "value": 1,
                      "valueType": "integer",
                      "cardinality": "one"
                    }
                  }
                ],
                "returnType": "boolean",
                "shortCircuit": true
              },
              "result": {
                "min": 10,
                "max": 65535,
                "description": "If SRT = 0 or 1 (scanned image data), HLL shall be an integer
from 10 through 65535.",
                "unit": "pixels",
                "scale": 0,
                "multipleOf": 1,
                "rounding": "exact"
              }
            }
          ],
          "description": "Logical horizontal line length constraint depends on the signature
representation type in 8.004:SRT."
        }
      },
      "representations": {
```

```
        "traditional": {
          "syntax": {
            "wireType": "binary",
            "contentEncoding": "raw-binary",
            "characterType": "Binary (B)",
            "format": "uint16",
            "length": {
              "fixed": 2,
              "description": "Traditional HLL is a two-byte unsigned binary integer."
            },
            "lengthUnit": "bytes",
            "notes": "Render the 2-byte binary value as hexadecimal. Display pattern: 0xHHHH.",
            "description": "Traditional Type-8 HLL is serialized as one unsigned two-byte
binary integer."
          }
        },
        "niem-xml-4": {
          "paths": [
            {
              "path": "itl:PackageSignatureImageRecord/itl:SignatureImage",
              "minCount": 1,
              "maxCount": 1
            },
            {
              "path":
"itl:PackageSignatureImageRecord/itl:SignatureImage/biom:ImageHorizontalLineLengthPixelQuantity
",
              "minCount": 1,
              "maxCount": 1
            }
          ],
          "syntax": {
            "wireType": "string",
            "contentEncoding": "none",
            "characterType": "Numeric (N)",
            "length": {
              "min": 1,
              "description": "Other HLL is represented as a one- to five-digit integer.",
              "max": 5
            },
            "lengthUnit": "characters",
            "format": [
              "N",
              "NN",
              "NNN",
              "NNNN",
              "NNNNN"
            ],
            "description": "NIEM XML HLL is serialized as one to five decimal numeric
characters. The value is 0 for vector data, otherwise 10 through 65535."
          }
        }
      }
    },
```


Fig. 8. Example JSON data for field 8.006