



AEROSPACE STANDARD

AS5684™**REV. B**

Issued 2008-09
Revised 2015-08
Reaffirmed 2025-04

Superseding AS5684A

(R) JAUS Service Interface Definition Language

RATIONALE

This document supersedes the Revision A publication of AS5684. This revision adds the following backwards compatible features

- Several new units are supported.
- `type_and_units_enum` is made reusable.
- Clarification for `default_state` transitions and `default_transitions` is provided.
- Support for entry and exit actions in derived services is added.

In addition, several ambiguities are addressed as well as updates to the RelaxNG schema.

AS5684B has been reaffirmed to comply with the SAE Five-Year Review policy.

INTRODUCTION

Services and service oriented architectures (SOAs) are widely used in the IT domain. The core principle of SOAs is the design of systems by means of distributed capabilities that are self-contained, loosely coupled, and have well defined interfaces. A service is a unit of work done by a service provider to achieve desired end results for a service consumer. Both provider and consumer are roles played by software agents on behalf of their owners. This principle fits very well in the unmanned systems domain. The JAUS Service Interface Definition Language (JSIDL) is a top-level specification that defines a language for describing unmanned system capabilities. It uses the Relax NG Compact [\[rng\]](#) syntax as a means to formally specify a schema that is to be used to create JAUS Service Definitions (JSDs) for all defined capabilities. Each JSD must be written in XML and must be validated against the JSIDL schema. The machine readable schema permits the development and use of automated tools, and consequently reduces integration time and provides for technology insertion.

This document is intended for anyone who wants to build or test JAUS Service Definitions. It is also intended for anyone who wants to build tools for designing, testing, documenting and/or implementing JAUS Service Definitions. The JAUS Service Definitions are intended for those who want to build or test a system that uses JAUS specifications.

SAE Executive Standards Committee Rules provide that: "This report is published by SAE to advance the state of technical and engineering sciences. The use of this report is entirely voluntary, and its applicability and suitability for any particular use, including any patent infringement arising therefrom, is the sole responsibility of the user."

SAE reviews each technical report at least every five years at which time it may be revised, reaffirmed, stabilized, or cancelled. SAE invites your written comments and suggestions.

Copyright © 2025 SAE International

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, or used for text and data mining, AI training, or similar technologies, without the prior written permission of SAE.

TO PLACE A DOCUMENT ORDER:
Tel: 877-606-7323 (inside USA and Canada)
Tel: +1 724-776-4970 (outside USA)
Fax: 724-776-0790
Email: CustomerService@sae.org
<http://www.sae.org>

SAE WEB ADDRESS:

Full document details: [SAE AS5684B](#)

For more information on this standard, visit
<https://www.sae.org/standards/content/AS5684B/>

Documents

Standard, visit Sharing Hub

TABLE OF CONTENTS

1.	SCOPE	4
1.1	J AUS Document Organization	4
2.	REFERENCES	4
2.1	SAE Publications	5
2.2	Applicable Documents	5
3.	LIST OF ACRONYMS	5
3.1	Definitions	6
3.1.1	Identifier (id)	6
3.1.2	Distributed System	6
3.1.3	Service	6
3.1.4	Service Definition	6
4.	FRAMEWORK FOR DEFINING PROTOCOLS / SERVICE DEFINITIONS	6
5.	ATTRIBUTES AND ELEMENTS OF A JAUS SERVICE DEFINITION	7
6.	MESSAGE ENCODING	9
6.1	Composite Fields	11
6.2	Simple Fields	18
6.2.1	Primitive Fields	18
6.2.2	String Fields	20
6.2.3	BLOB Fields	21
6.3	Meta Fields	22
6.4	Field Information Attributes and Elements	23
6.4.1	Field Information Attributes	24
6.4.2	Field Information Elements	25
6.5	Declared Types and Constants	27
7.	BEHAVIOR	32
7.1	Separation of Protocol Behavior from Application Behavior	32
7.2	Protocol Behavior	33
7.3	States	35
7.3.1	Entry and Exit Actions	35
7.4	Transitions	37
7.4.1	Simple Transitions	38
7.4.2	Loop-back Transitions (Deprecated)	39
7.4.3	Internal Transitions	39
7.4.4	Push Transitions	40
7.4.5	Pop Transitions	40
7.5	Default States and Default Transitions	41
8.	SERVICE REFERENCE RELATIONSHIPS	42
8.1	Inheritance	43
8.2	Client	46
9.	AN EXAMPLE OF A SERVICE DEFINITION	48
10.	VERSIONING AND BACKWARDS COMPATIBILITY	53
11.	COMPLIANCE	53
12.	NOTES	53
APPENDIX A	SCHEMA	54

Figure 1	Scope of JSIDL with reference to other frameworks and architectures	4
Figure 2	Top level structure of a service definition	7
Figure 3	Serialized version of a section of a JAUS message	9
Figure 4	Message_def - a three part structure	9
Figure 5	Data field elements - represent encoded data	10
Figure 6	Array - a homogenous sequence of simple fields	11
Figure 7	Record - a heterogeneous set of simple fields (including arrays) that is ordered	13
Figure 8	List - a variable sized sequence of simple or composite fields of the same type	14
Figure 9	Variant - a container of run-time selectable types	14
Figure 10	An empty variant - creates the possibility of specifying no data	15
Figure 11	Sequence - a heterogeneous sequence of composite fields (excluding arrays)	16
Figure 12	Example of a fixed-size/shape N-ARY tree	16
Figure 13	Example of an adjacency list representation of a graph	17
Figure 14	Field information attributes and elements - provide information about encoded data	24
Figure 15	Example of a reference to a declared_type	29
Figure 16	Separation of protocol behavior from application behavior	33
Figure 17	Structure of the protocol_behavior element	34
Figure 18	The inherits-from element	43
Figure 19	Addition of features - nested states	44
Figure 20	Addition of features - concurrent state machines	44
Figure 21	Addition of features - transition	45
Figure 22	The client-of element	47
Figure 23	UML state diagram of the access control service	48
Figure 24	UML state diagram of the extended access control service	51
Table 1	Size and representation of primitive data types	24
Table 2	Types of transitions for which entry/exit actions are executed [SMC]	36

1. SCOPE

The SAE Aerospace Information Report AIR5315 – Generic Open Architecture (GOA) defines “a framework to identify interface classes for applying open systems to the design of a specific hardware/software system.” [\[sae\]](#) JAUS Service (Interface) Definition Language defines an XML [schema](#) for the interface definition of services at the Class 4L, or Application Layer, and Class 3L, or System Services Layer, of the Generic Open Architecture stack (see Figure 1). The specification of JAUS services shall be defined according to the JAUS Service (Interface) Definition Language document.

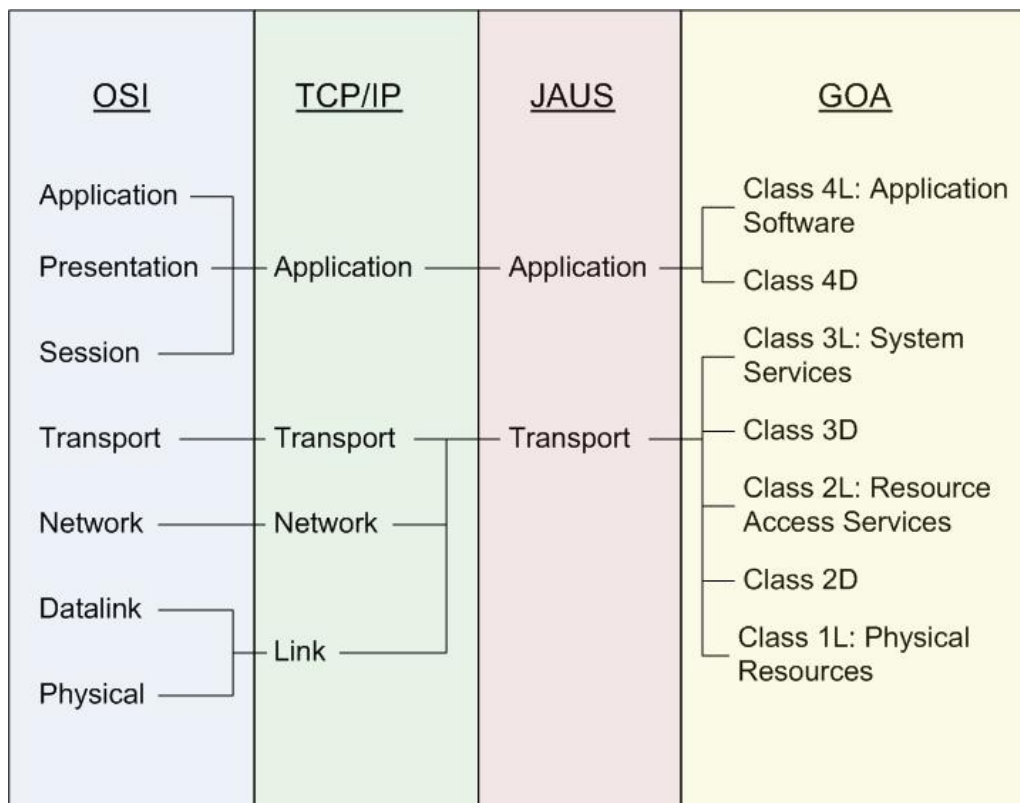


Figure 1 - Scope of JSIDL with reference to other frameworks and architectures

1.1 JAUS Document Organization

The layout of this document is as follows. [Section 3](#) lists external references. [Section 4](#) and [Section 5](#) describe the elements of a JAUS Service Definition: description, assumptions, message set, message encoding and protocol behavior. The bulk of the JAUS Service Definition lies in the last two elements, Message Encoding and Behavior. These two elements are described in detail in [Section 6](#) and [Section 7](#). [Section 8](#) describes two service reference relationships that allow for the reuse of existing service definitions. [Section 9](#) contains an example of a JAUS Service Definition. Version control rules for the JSIDL and all Service Definitions are presented in [Section 10](#). [Section 11](#) contains a short note on compliance. [Appendix A](#) contains the complete JSIDL.

Almost all figures in the document use UML 2.0 notation [\[uml\]](#).

2. REFERENCES

The following publications form a part of this document to the extent specified herein. The latest issue of SAE publications shall apply. The applicable issue of other publications shall be the issue in effect on the date of the purchase order. In the event of conflict between the text of this document and references cited herein, the text of this document takes precedence. Nothing in this document, however, supersedes applicable laws and regulations unless a specific exemption has been obtained.

2.1 SAE Publications

Available from SAE International, 400 Commonwealth Drive, Warrendale, PA 15096-0001, Tel: 877-606-7323 (inside USA and Canada) or +1 724-776-4970 (outside USA), www.sae.org.

AIR5315 Overview and Rationale for GOA Framework Standard

AS5669 JAUS/SDP Transport Specification

2.2 Applicable Documents

[ansic] Brian W. Kernighan. Dennis M. Ritchie. The C Programming Language (ANSI C). Prentice Hall. 1997.

[bnt] Barry N. Taylor. The International System of Units (SI), National Institute of Standards and Technology Special Publication 330, 1991 Edition. <http://physics.nist.gov/Document/sp330.pdf>

[greg] John D. McGregor, Douglas M. Dyer, A Note on Inheritance and State Machines, ACM SIGSOFT, Software Engineering Notes vol. 18 no. 4, Oct 1993 p. 61.

[holz] Gerard J. Holzmann. Design and Validation of Computer Protocols. Prentice Hall Software Series. 1991.

[lsp] Barbara Liskov. Data abstraction and hierarchy. Conference on Object Oriented Programming Systems Languages and Applications. pp. 17 – 34, 1987.

[lynch] Nancy A. Lynch. Distributed Algorithms. Morgan Kaufmann Publishers. April, 1997.

[rng] Relax NG: <http://www.oasis-open.org/committees/relax-ng/spec-20011203.html>, Standard lightweight XML schema language.

[smc] State Machine Compiler (SMC), <http://smc.sourceforge.net/>.

[uml] Unified Modeling Language (UML), version 2.1.1, formal/2007-02-05, <http://www.omg.org/docs/formal/07-02-05.pdf>

[uri] Uniform Resource Identifiers (URI): Generic Syntax, <http://ftp.ics.uci.edu/pub/ietf/uri/rfc2396.txt>.

[uuid] A Universally Unique Identifier (UUID) URN Namespace, <http://www.ietf.org/rfc/rfc4122.txt>.

[xml] Extensible Markup Language (XML), <http://www.w3.org/XML/>.

3. LIST OF ACRONYMS

ANSI	American National Standards Institute
Id	Identifier
API	Application Programming Interface
UTF-8	UCS (Unicode) Transformation Format-8
BLOB	Binary Large Object
DOM	Document Object Model
IEEE	Institute of Electrical and Electronics Engineers
JAUS	Joint Architecture for Unmanned Systems

JSD	J AUS Service Definition
JSIDL	J AUS Service (Interface) Definition Language
NIST	National Institute of Standards and Technology
SMC	State Machine Compiler
UML	Unified Modeling Language
URL	Uniform Resource Locator
URN	Uniform Resource Name
URI	Uniform Resource Identifier
UUID	Universally Unique Identifier
XML	Extensible Markup Language

3.1 Definitions

3.1.1 IDENTIFIER (id)

An identifier is an object that can act as a reference to something that has identity [\[uri\]](#).

A basic requirement for identifiers is that they must be unique within the scope in which they are defined.

3.1.2 DISTRIBUTED SYSTEM

A Distributed System is one in which the power of several computing entities is aggregated to collaboratively run a single computational task in a transparent and coherent way, so that they appear as a single, centralized system.

3.1.3 SERVICE

A service is a unit of work done by a service provider to achieve desired end results for a service consumer.

3.1.4 SERVICE DEFINITION

The definition of a service (or service interface) comprises of an interface to a coherent function. A service definition specifies only the messaging syntax, semantics, and protocol necessary to use the function. It does not specify how the function is implemented.

4. FRAMEWORK FOR DEFINING PROTOCOLS / SERVICE DEFINITIONS

This specification employs the framework for protocol definition developed by Holzmann. In this specification, the term “service” is equivalent to Holzmann’s use of the term “protocol.” In outline form, a complete and unambiguous specification of a protocol requires:

“A protocol specification consists of five distinct parts. To be complete, each specification should include explicitly:

- The service to be provided by the protocol
- The assumptions about the environment in which the protocol is executed
- The vocabulary of messages used to implement the protocol

- d. The encoding (format) of each message in the vocabulary
- e. The procedure rules guarding the consistency of message exchanges

The fifth element of a protocol specification is the most difficult to design” [holz p. 21]

The JAUS Service (Interface) Definition Language provides a means for defining each of these five elements.

5. ATTRIBUTES AND ELEMENTS OF A JAUS SERVICE DEFINITION

This section describes the elements of a JAUS Service Definition. The actual schema is presented in [Appendix A](#). Figure 2 illustrates the top-level structure of a service definition. The root element of the definition is `service_def`. All the attributes and elements of a `service_def` are required, except for the [declared_type_set](#) element which is optional. All JAUS Service Definitions must be specified under a JSIDL namespace Uniform Resource Identifier (or URI) [\[uri\]](#).

```
<service_def
  xmlns="urn:jaus:jsidl:0.11"
  name="service_name"
  id="service_uri"
  version="1.1"
  deprecated="false">

  <description> </description>

  <assumptions> </assumptions>

  <references> </references>

  <declared_const_set> </declared_const_set>
  <declared_type_set> </declared_type_set>

  <message_set>
    <input_set> </input_set>
    <output_set> </output_set>
  </message_set>

  <internal_events_set> </internal_events_set>

  <protocol behavior> </protocol behavior>
```

Figure 2 - Top level structure of a service definition

Attributes

- **name:** The name of the service. The name must be a valid identifier (refer Appendix A.4): It must also be unique among all service names used within the scope of the namespace in which the service is specified.
- **id:** The namespace of a service is defined by its id. The id is a globally unique URI that uniquely identifies a particular service. All versions of a particular service must have the same id. The chosen URI can be either a Uniform Resource Locator (URL) or a Uniform Resource Name (URN). The choice will depend on whether a location dependent id is preferred or a location independent id is preferred. For example, an organization that does not have or want to publish and maintain a web page for each service would prefer to use URNs over URLs. For services that are either experimental or designed for temporary use, a particular type of URN called a Universally Unique Identifier (UUID) [\[uuid\]](#) may be more applicable. UUIDs are 16 byte identifiers that are generated algorithmically and are globally unique in space and time.

- version: The version number of a service. Details about a service's version number is provided in [Section 10: Versioning and Backwards Compatibility](#).
- deprecated: An optional Boolean (true, false) flag indicating that the service has been deprecated and is scheduled to be removed in a future publication.

Elements

- description: A required element that must be used to provide a brief textual description of the service and its functionality. The description element has an attribute called xml:space. Basic textual formatting can be preserved in the description element if the value of this attribute is set to “preserve”.
- assumptions: A required element that must be used to provide a brief textual description about the assumptions made about the nature of the communication channels the service uses to communicate with its clients.
- references: An optional element that can be used to provide a list of service definitions that the service references. [Section 8](#) describes service reference relationships in detail.
- declared_const_set: An optional element that can be used to define a reusable set of constant definitions (see [6.5](#)).
- declared_type_set: An optional element that can be used to define a reusable set of data type definitions (see [6.5](#)).
- message_set: A required element that must be used to define the input and output message sets that make up the external vocabulary of the service. When a service definition references other service definitions, the vocabulary of the service is determined by the nature of the reference used (see [Section 8](#)). Each message definition within the message_set contains a detailed encoding of the structure of the message data. [Section 6](#) provides a detailed description on encoding, while [Appendix A.2](#) contains the schema.
- Internal_events_set: The internal events set is used to define events¹ like timeouts that occur internal to a service and have a direct effect on the protocol behavior of the service. Like message definitions, event definitions can contain a detailed specification of the encoding of data that may be associated with the event. [Section 6](#) provides a detailed description on encoding, while [Appendix A.2](#) contains the schema.
- protocol_behavior: The protocol or message exchange behavior of a service is defined in the form of a finite state machine. The definition of a service is structured around the separation of the behavior that is related to the exchange of messages (protocol) and the application behavior application behavior (see [Section 7](#) on separation of protocol and application behavior). The behavior specified by the service definition is that of the protocol and not the application. For example, the behavior element of a waypoint driver service definition must describe only the message interchange behavior between the service and its clients. It should not specify the internal computation required to say, generate wrench commands used to reach a given waypoint. A detailed description on specifying service behavior is provided in [Section 7](#), while [Appendix A.3](#) contains the schema for specifying protocol behavior.

¹ The structure of the event_def element is similar to the structure of a message_def element except that it does not have message_id and is_command attributes.

6. MESSAGE ENCODING

The format of a JAUS message is defined using a `message_def` element. Although the message structure and format is specified in XML, the format of the actual on-the-wire JAUS message is a binary format containing only the serialized data. Figure 3 shows a section of a serialized JAUS message. This message contains a sequence of twenty bytes. The first two bytes represent a short integer that is interpreted as the waypoint number. The next twelve bytes represent a sequence of three 4-byte integers interpreted as latitude, longitude and elevation. The last six bytes represent three short integers that are interpreted as the roll, pitch and yaw. For inter-process communication, all multi-byte data like the short integer and 4-byte integer in a JAUS message that is not opaque data (like a JPEG image for example) is encoded in little endian where the low byte comes first. As an exception to this rule, the native format of the processor may be used for encoding messages that are used for communicating data within the same processor.

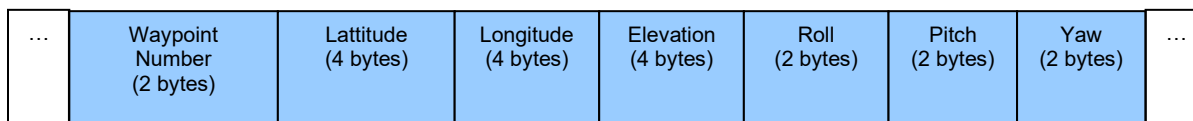


Figure 3 - Serialized version of a section of a JAUS message

The specification of such a format using the `message_def` element includes all the information necessary to format and interpret the message unambiguously. The `message_def` element has a three part structure: header, body and footer, each of which may be specified as being empty. It also contains an element called description that may be used to provide a brief textual description of the message. The description element provides a basic white space formatting capability through its attribute called `xml:space`. If this attribute is assigned the value “default”, then XML applications (readers and editors) will not adhere to the white space formatting that may have been used. If this attribute is assigned the value “preserve”, then XML applications will adhere to the white space formatting that may have been used in the description element. The `message_def` element has attributes for specifying the message name, a message id (in hexadecimal), an optional boolean value that specifies whether the message contains a command within it, and an optional value that specifies whether the message is deprecated and scheduled to be removed in a later publication. The message ids assigned to messages belonging to the input and output set are globally unique.

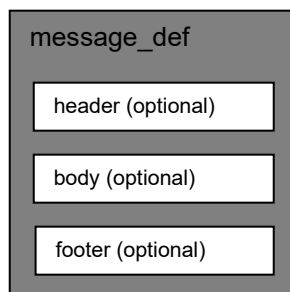


Figure 4 - Message_def - a three part structure

The JSIDL provides several elements and attributes for defining the structure and content of the header, body and footer of a message. These elements and attributes are broadly classified into two groups: 1) Data Field Elements, and 2) Field Information Elements and Attributes. Data Field Elements represent encoded data and are further divided into three groups – Composite Fields, Simple Fields and Meta Fields. These groups are illustrated in Figure 5 and defined in [6.1](#), [6.2](#) and [6.3](#) respectively. These fields have been designed for bandwidth optimization and to provide flexibility to the designer of Service Definitions. [Section 6.4](#) contains the definitions of the Field Information Elements and Attributes. These elements and attributes are used in data field definitions to provide information about the encoded data. Finally, [6.5](#) describes the specification of messages and their contents as declared types for the purpose of reuse.

Examples:

```

<message_def name="Request_Control" message_id="000d">
  <description xml:space="preserve">
    This message is used to request exclusive control of...
  </description>
  <header>
    ...
  </header>
  <body>
    ...
  </body>
  <footer/>
</message_def>

```

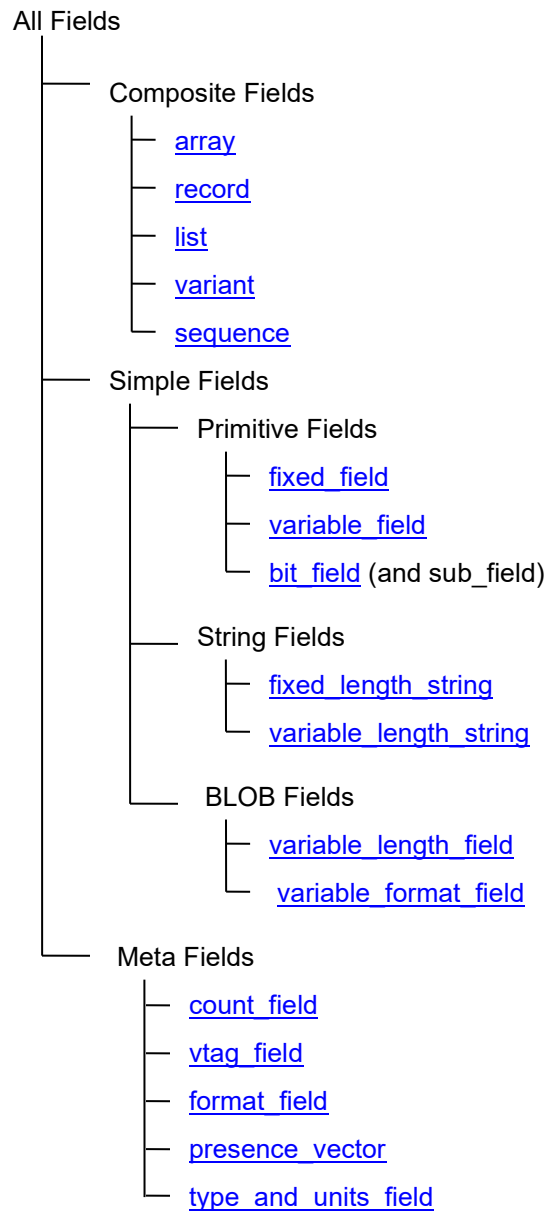


Figure 5 - Data field elements - represent encoded data

6.1 Composite Fields

Composite Fields are containers that provide a means to create complex data structures by combining a set of [Simple Fields](#) in various ways. There are five types of complex fields: array, record, list, variant and sequence. Arrays can only be contained by records and records can only be contained by lists, variants and sequences. In addition to records, lists, variants and sequences can contain lists, variants and sequences.

array

An array field represents a rectangular multi-dimensional array composed of [Simple Fields](#) of the same type. Its structure is illustrated in Figure 6. The definition for an array must contain a Simple Field template for the elements of the array. It must also contain definitions for the dimensions of the array.

The array has a name attribute that is required, and an optional interpretation attribute that may be used to provide a brief textual interpretation for the array. The name of the array must be a valid identifier (refer [Appendix A.4](#)) and must be unique within the scope of the encapsulating [record](#) element. The dimensions of the array are specified using a sub-element called dimension. Each dimension definition consists of the name that is attributed to the dimension, the size of the dimension, and a brief textual interpretation that is optional. The name of the dimension must be a valid identifier (refer [Appendix A.4](#)) and must be unique within the scope of the array. The size of the dimension must be an integer constant or a reference to an integer constant (see [6.5](#)).

The order in which the dimensions are specified indicates how the array is read. If the order of the dimensions of an m -dimensional array is $d_1 \times d_2 \times \dots \times d_{m-1} \times d_m$, it implies that the subarray $d_1 \times d_2 \times \dots \times d_{m-1}$ is repeated d_m times. When applied recursively, $d_1 \times d_2 \times \dots \times d_{m-2}$ is repeated $d_{m-1} \times d_m$ times. Each two-dimensional sub-array is read row first.

An array defined within a [record](#) can be optional at runtime. This is specified by the boolean attribute called optional. If this attribute is set to true, the entire array may not be included in the record if it is not required (see definition of [record](#)). Note that the 'optional' attribute of a Simple Field defined as the element in an array is ignored when used within the array.

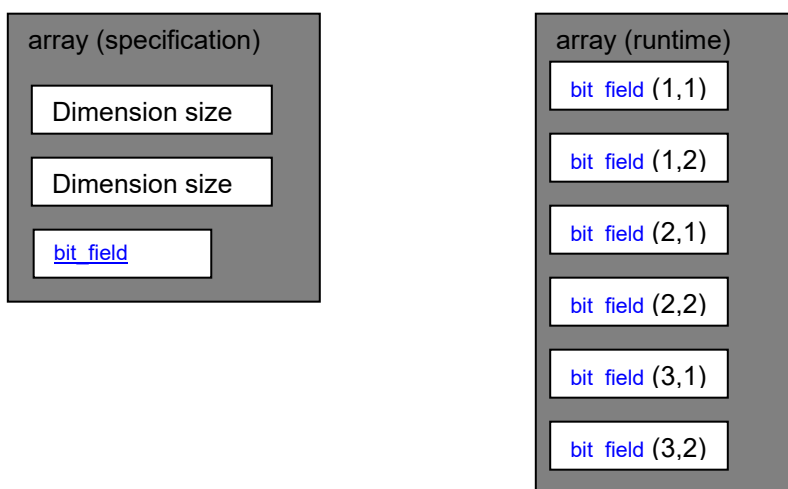


Figure 6 - Array - a homogenous sequence of simple fields

Examples:

```
<array name="Raster_Data" optional="false" interpretation="A 2-Dimensional scan....">
  <!--the semantics of the optional attribute of pixel is undefined -->
  <fixed_field name="Pixel" field_type="byte" field_units="one" optional="false" interpretation="A
single pixel value"/>
  <dimension name="RGB" size="3" interpretation="Red, Blue and Green Values"/>
  <dimension name="Color_Vector" size="1" interpretation="RGB Color vector"/>
  <dimension name="width" size="100"/>
  <dimension name="height" size="100"/>
</array>
```

This stands for a color vector repeated 100 x 100 times. Each 3 x 1 color vector contains three unsigned bytes standing for red, green and blue values².

The serialization of this array to obtain the on-the-wire format as shown step-wise below.

When the first two-dimensional array is serialized, we get

{r_1, g_1, b_1}.

In the 100x100 matrix, this represents row1, column1.

Adding the next dimension we get a new two-dimensional array, ((r_1, g_1, b_1), 100) which when serialized yields,

{r_1, g_1, b_1, r_2, g_2, b_2, ..., r_100, g_100, b_100}.

In the 100x100 matrix, this represents the entire row1.

Adding the next dimension we get a new two-dimensional array,

((r_1, g_1, b_1, r_2, g_2, b_2, ..., r_100, g_100, b_100), 100)

which when serialized yields,

{r_1, g_1, b_1, r_2, g_2, b_2, ..., r_100, g_100, b_100, ..., r_10000, g_10000, b_10000}.

In the 100x100 matrix, this represents rows 1 to 100.

record

A record represents an arrangement of one or more [Simple Fields](#) or [arrays](#) leading to a heterogeneous or homogeneous set that is ordered. Its structure is illustrated in Figure 7.

The Simple Fields and arrays contained within a record can be optional at runtime. That is, they may or may not be included in the message. If any of the Simple Fields or arrays within a record are specified as being optional, the record must have a Meta Field called the [presence vector](#) as its first field. This Meta Field is used to indicate the presence or absence of the optional Simple Fields or arrays within the record.

The record has a name attribute that is required, and an optional interpretation attribute that may be used to provide a brief textual interpretation for the record. The name of the record must be a valid identifier (refer [Appendix A.4](#)) and must be unique within the scope of the encapsulating [body](#) or [list](#) parent element.

A record defined within a sequence can be optional at runtime. This is specified by the boolean attribute called optional. If this attribute is set to true, the entire record may not be included in the sequence if it is not required (see definition of sequence).

² Note that a dimension of size 1 has no effect on the serialization of the array. Its purpose is to clearly define (or identify) elements and sub-elements of the array as vectors.

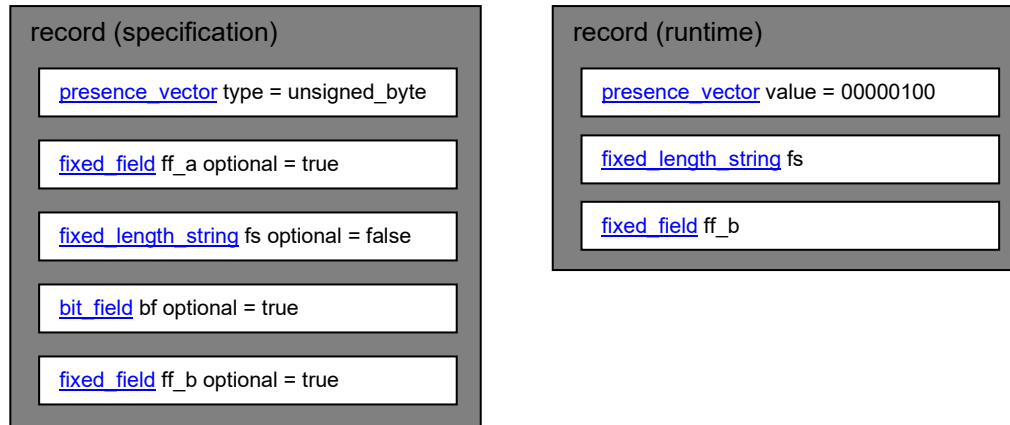


Figure 7 - Record - a heterogeneous set of simple fields (including arrays) that is ordered

Examples:

```
<!-- A record with a single Simple Field. -->
<record name="a_record">
  <fixed_field>...</fixed_field>
</record>

<!-- A record containing a presence vector followed by five
Simple Fields, some or all of which are optional (since
the presence vector is present). -->
<record name="some_record">
  <presence_vector field_type_unsigned="unsigned_byte"/>
  <fixed_field ...> ... </fixed_field>
  <fixed_field ...> ... </fixed_field>
  <fixed_length_string ...optional="true">
    ...
  </fixed_length_string>
  <fixed_field ...> ... </fixed_field>
  <variable_field ...optional="true">
    <array...>...</array>
  </variable_field>
  <fixed_field ...> ... </fixed_field>
</record>
```

list

A list field represents a variable sized sequence of Composite fields of the same type, with the exception of array. The size of the sequence is determined at runtime from the value a Meta field called `count_field` which is the first field in the list.

A list defined within a sequence can be optional at runtime. This is specified by the boolean attribute called `optional`. If this attribute is set to `true`, the entire list may not be included in the sequence if it is not required (see definition of sequence). Note that the 'optional' attribute of a Composite Field defined as a top-level element in a list is ignored when used within the list. Any optional fields contained within a top-level element Composite Field are not impacted by this.

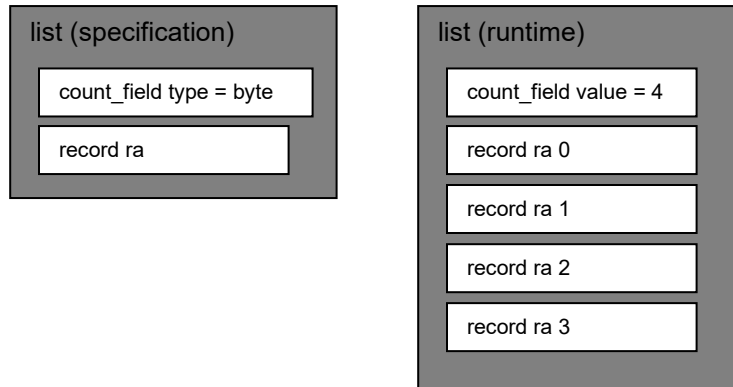


Figure 8 - List - a variable sized sequence of composite fields of the same type

Examples:

```
<list name="subsystyem_list" interpretation="list of subsystems in the system">
  <count_field field_type_unsigned="unsigned byte"/>
  <record name="subsystem_rec" interpretation="name,id pair">
    <fixed_field name="subsystem_id" field_type="unsigned byte" field_units="one"
optional="false"/>
    <fixed_length_string name="subsystem_name" string_length="80" optional="false"/>
  </record>
</list>
```

variant

A variant field represents a composite field that can hold zero or one of several different types of pre-defined composite fields at runtime. The pre-defined set of composite fields are specified as a sequence and the type of field contained within the variant at runtime is determined from the value a Meta field called the vtag_field which is the first field in the variant. If the n^{th} composite field in the sequence is chosen, then the value of the vtag_field must be set to n , where $n=0$ if the first composite field in the sequence is chosen at runtime.

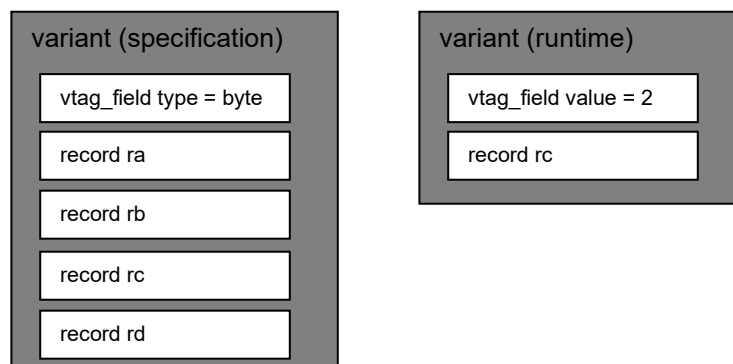


Figure 9 - Variant - a container of run-time selectable types

A variant may be specified as being empty. That is, it does not specify a sequence of composite fields (or the sequence is of size zero). This case is useful when one of the possible choices of a variant has no data at all. This can be specified by including an empty variant within a variant. In this case, the vtag field of the empty variant must be set to 0 as shown in Figure 10.

A variant defined within a sequence can be optional at runtime. This is specified by the boolean attribute called optional. If this attribute is set to true, the entire variant may not be included in the sequence if it is not required (see definition of sequence). Note that the 'optional' attribute of a Composite Field defined as a top-level element in a variant is ignored when used within the variant. Any optional fields contained within a top-level element Composite Field are not impacted by this.

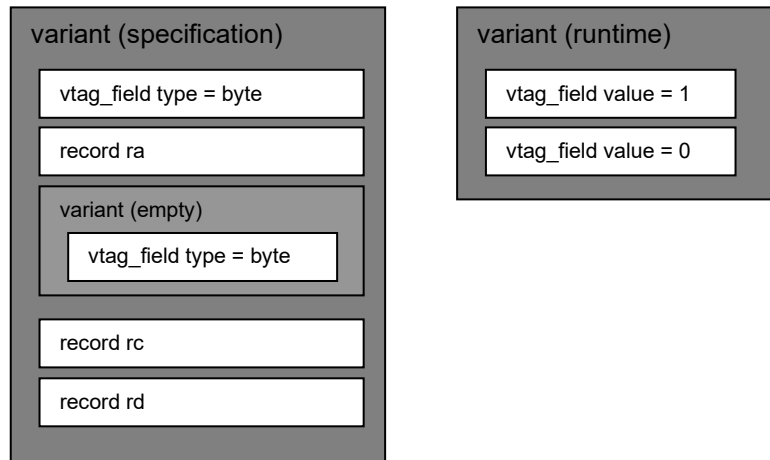


Figure 10 - An empty variant - creates the possibility of specifying no data

Examples:

```
<variant name="shape">
  <vtag_field field_type_unsigned="unsigned byte" min_count="0" max_count="2"/>
  <record name="rectangle">
    <fixed_field name="length" field_type="byte" field_units="one" optional="false"/>
    <fixed_field name="height" field_type="byte" field_units="one" optional="false"/>
  </record>
  <record name="square">
    <fixed_field name="side" field_type="byte" field_units="one" optional="false"/>
  </record>
  <record name="circle">
    <fixed_field name="radius" field_type="byte" field_units="one" optional="false"/>
  </record>
</variant>
```

sequence

As the name suggests, a sequence represents an arrangement of one or more Composite Fields (excluding [arrays](#)) leading to a heterogeneous or homogeneous sequence. Its structure is illustrated in Figure 11.

The Composite Fields contained within a sequence can be optional at runtime. That is, they may or may not be included in the message. If any of the composite fields within a sequence are specified as being optional, the sequence must have a Meta Field called the [presence vector](#) as its first field. This Meta Field is used to indicate the presence or absence of the optional Composite Fields within the record.

A sequence defined within another sequence can be optional at runtime. This is specified by the boolean attribute called optional. If this attribute is set to true, the entire encapsulated sequence may not be included in the container sequence if it is not required.

The recursive structure of list, variant and sequence fields can be used to define data structures like graphs and trees. Examples of these structures are presented in Figures 12 and 13.

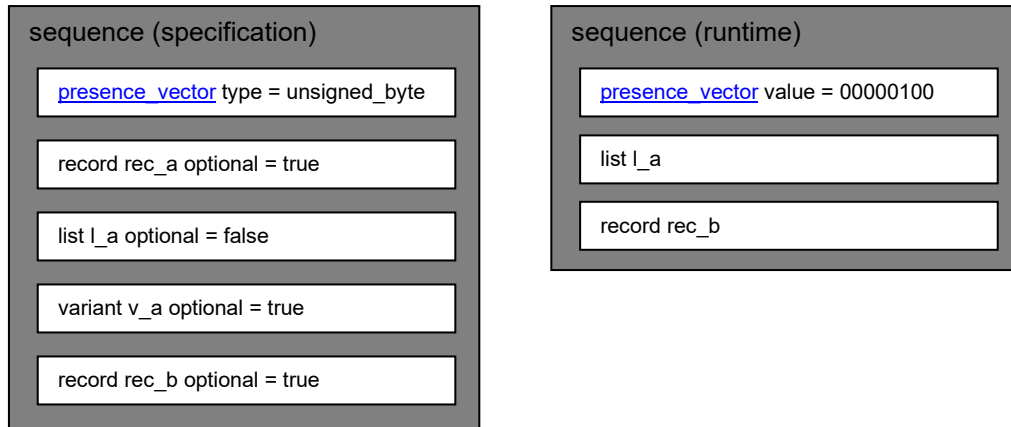


Figure 11 - Sequence - a heterogeneous sequence of composite fields (excluding arrays)

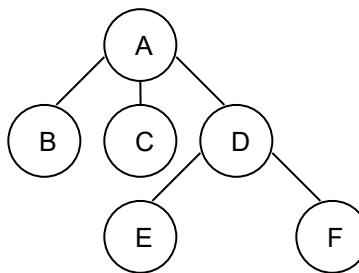


Figure 12 - Example of a fixed-size/shape N-ARY tree

Examples:

```

<sequence name="A">
  <record name="a">
    <fixed_field name="a" field_type="byte" field_units="one" optional="false"/>
    <fixed_field name="b" field_type="byte" field_units="one" optional="false"/>
  </record>
  <sequence name="B">
    <record name="b">
      <fixed_field name="b" field_type="byte" field_units="one" optional="false"/>
    </record>
  </sequence>
  <sequence name="C">
    <record name="c">
      <fixed_field name="c" field_type="byte" field_units="one" optional="false"/>
    </record>
  </sequence>
</sequence>

```

```

<sequence name="D">
  <sequence name="E">
    <record name="e">
      <fixed_field name="e" field_type="byte" field_units="one" optional="false"/>
    </record>
  </sequence>
</sequence>
<sequence name="F">
  <record name="f">
    <presence_vector field_type_unsigned="unsigned byte"/>
    <fixed_field name="f" field_type="byte" field_units="one" optional="true"/>
    <fixed_field name="g" field_type="byte" field_units="one" optional="true"/>
  </record>
</sequence>
</sequence>
</sequence>

```

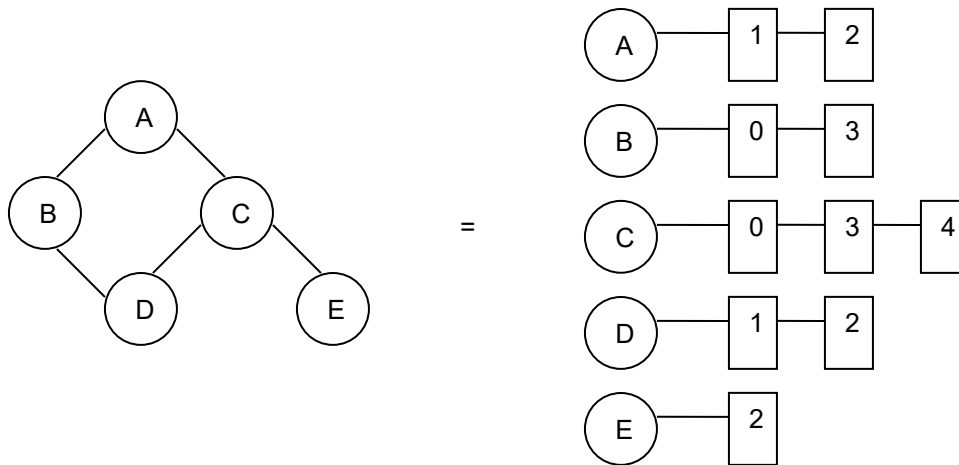


Figure 13 - Example of an adjacency list representation of a graph

```

<sequence name="adjacencyListGraph">
  <sequence name="A">
    <record name="a">
      <fixed_field name="a" field_type="byte" field_units="one" optional="false"/>
      <fixed_field name="b" field_type="byte" field_units="one" optional="false"/>
    </record>
    <list name="edges">
      <count_field field_type_unsigned="unsigned byte"/>
      <record name="index_rec">
        <fixed_field name="index" field_type="byte" field_units="one" optional="false"/>
      </record>
    </list>
  </sequence>
  <sequence name="B">
    <record name="b">
      <fixed_field name="b" field_type="byte" field_units="one" optional="false"/>
    </record>
    <list name="edges">
      <count_field field_type_unsigned="unsigned byte"/>
      <record name="index_rec">
        <fixed_field name="index" field_type="byte" field_units="one" optional="false"/>
      </record>
    </list>
  </sequence>
  <sequence name="C">
    <record name="c">
      <fixed_field name="c" field_type="byte" field_units="one" optional="false"/>
    </record>
  </sequence>

```

```

<list name="edges">
  <count_field field_type_unsigned="unsigned byte"/>
  <record name="index_rec">
    <fixed_field name="index" field_type="byte" field_units="one" optional="false"/>
  </record>
</list>
</sequence>
<sequence name="D">
  <record name="d">
    <fixed_field name="d" field_type="byte" field_units="one" optional="false"/>
  </record>
  <list name="edges">
    <count_field field_type_unsigned="unsigned byte"/>
    <record name="index_rec">
      <fixed_field name="index" field_type="byte" field_units="one" optional="false"/>
    </record>
  </list>
</sequence>
<sequence name="E">
  <record name="e">
    <fixed_field name="e" field_type="byte" field_units="one" optional="false"/>
  </record>
  <list name="edges">
    <count_field field_type_unsigned="unsigned byte"/>
    <record name="f">
      <fixed_field name="index" field_type="byte" field_units="one" optional="false"/>
    </record>
  </list>
</sequence>
</sequence>

```

<list name="variable_size_shape_n_ary_tree" interpretation="A variable size and shape n-ary tree and graph. All nodes in this graph hold the same type of data. But, this is not necessary. Nodes can be heterogeneous.">

```

  <count_field field_type_unsigned="unsigned byte" interpretation="node count"/>
  <sequence name="node">
    <record name="node_data">
      <fixed_field name="data" field_type="byte" optional="false" field_units="one"/>
    </record>
    <list name="edges">
      <count_field field_type_unsigned="unsigned byte" interpretation="edge count"/>
      <record name="edge_index">
        <fixed_field name="edge_index" field_type="byte" field_units="one" optional="false"/>
      </record>
    </list>
  </sequence>
</list>

```

6.2 Simple Fields

A Simple Field is an abstract data type that must be used to specify a single cohesive unit of data that cannot be further divided into two or more Simple or [Composite Fields](#). Simple Fields may contain one or more [Meta Fields](#). There are three kinds of Simple Fields: Primitive Fields, [String Fields](#) and [Binary Large Object \(BLOB\) Fields](#).

6.2.1 Primitive Fields

Primitive Fields represent data with primitive types such as byte, integer, float etc. These types are described in the Field Information attribute [field_type](#).

fixed_field

A `fixed_field` represents a primitive data type whose type and units are fixed. The type of the `fixed_field` is specified by the [field_type](#) attribute and its units are specified by the [field_units](#) attribute.

The default value range of a `fixed_field` is the range of the chosen `field_type`. If the values of a `fixed_field` needs to be further constrained, the `child` element must be used.

If the `fixed_field` represents a scaled integer, the range for the scale must be specified with the [scale_range](#) child element.

A `fixed_field` used within a [record](#) can be optional at runtime. This is specified by setting the attribute called `optional` to `true`, else it must be set to `false`.

The name of the `fixed_field` must be specified using its `name` attribute. The name must be a valid identifier (refer [Appendix A.4](#)) and it must be unique within the scope of the encapsulating record. The `interpretation` attribute is an optional attribute that may be used to provide a textual description of the `fixed_field`.

Examples:

```
<fixed_field field_type="short integer" field_units="radian" name="Roll" optional="true">
  <scale_range real_lower_limit="-PI" real_upper_limit="PI" integer_function="round"/>
</fixed_field>
```

```
<fixed_field field_type="unsigned short integer" name="Waypoint_Number" field_units="one"
optional="false">
  <value_set offset_to_lower_limit="false">
    <value_range lower_limit="0" upper_limit="65535" lower_limit_type="inclusive"
      upper_limit_type="inclusive"/>
  </value_set>
</fixed_field>
```

```
<fixed_field name="priority" field_type="byte" field_units="one" optional="false">
  <value_set offset_to_lower_limit="false">
    <value_range lower_limit="0" upper_limit="9" lower_limit_type="inclusive"
      upper_limit_type="inclusive"/>
    <value_enum enum_index="0" enum_const="'low priority'" interpretation="implies a low
      priority message."/>
    <value_enum enum_index="5" enum_const="'medium priority'" interpretation="implies
      medium priority message."/>
    <value_enum enum_index="9" enum_const="'high priority'" interpretation="implies a
      high priority message."/>
  </value_set>
</fixed_field>
```

variable_field

A `variable_field` is a Simple Field whose type and units can vary at runtime. This field is always preceded by a Meta Field called [type_and_units_enum](#) that specifies its type and units.

A `variable_field` used within a [record](#) can be optional at runtime. This is specified by setting the attribute called `optional` to `true`, else it must be set to `false`.

The name of the `variable_field` must be specified using its `name` attribute. The name must be a valid identifier (refer [Appendix A.4](#)) and it must be unique within the scope of the encapsulating record. The `interpretation` attribute is an optional attribute that may be used to provide a textual description of the `variable_field`.

Examples:

```
<variable_field name="temperature" interpretation="CPU temperature" optional="false">
  <type_and_units_enum index="0" field_type="short integer" field_units="degree Celsius"/>
  <type_and_units_enum index="1" field_type="short integer" field_units="kelvin"/>
  <type_and_units_enum index="2" field_type="float" field_units="degree Celsius"/>
  <type_and_units_enum index="3" field_type="float" field_units="kelvin"/>
</variable_field>
```

bit_field

The `bit_field` is a Simple Field that can be used to specify one or more bit-level fields within a single [primitive data type](#) whose type must be specified by the [field_type](#) attribute.

Bit-level fields within a `bit_field` are specified by an ordered list of Field Information elements called [sub_field](#). The unspecified bits of a `bit_field` must always be set to 0.

A `bit_field` used within a [record](#) can be optional at runtime. This is specified by setting the attribute called `optional` to true, else it must be set to false.

The name of the `bit_field` must be specified using its `name` attribute. The name must be a valid identifier (refer [Appendix A.4](#)) and it must be unique within the scope of the encapsulating record. The `interpretation` attribute is an optional attribute that may be used to provide a textual description of the `bit_field`.

Examples:

```
<bit_field name="two_sub_fields" field_type_unsigned="unsigned byte" optional="false">
  <sub_field name="sub_1">
    <bit_range from_index="0" to_index="3"/>
    <value_set offset_to_lower_limit="false">
      <value_range lower_limit="0" upper_limit="12" lower_limit_type="inclusive"
upper_limit_type="inclusive"/>
    </value_set>
  </sub_field>
  <sub_field name="sub_2">
    <bit_range from_index="4" to_index="7"/>
    <value_set offset_to_lower_limit="false">
      <value_enum enum_index="0" enum_const="'str const 0'" interpretation="index holds the integer
value of the enum"/>
      <value_enum enum_index="1" enum_const="'str const 1'" interpretation="value holds a string
constant that maps to the index"/>
      <value_enum enum_index="2" enum_const="'str const 2'" />
    </value_set>
  </sub_field>
</bit_field>
```

sub_field

Each field contained within a [bit_field](#) is specified by a `sub_field` element. `sub_fields` use the [bit_range](#) element to specify the bits used for each field within a `bit_field`, and the [value_set](#) element to specify the values that each field can take.

The name of a `sub_field` must be specified by its `name` attribute and must be a valid identifier (refer [Appendix A.4](#)). `sub_field` also has an optional attribute for a textual interpretation of the specified field.

Examples: Refer [bit_field](#) examples

6.2.2 String Fields

String Fields represent character string data derived from the Unicode Transformation Format (UTF-8).

fixed_length_string

A `fixed_length_string` is a Simple Field that contains a character string whose maximum length is fixed. The maximum length of the string must be specified by an attribute called `string_length` whose value must be an integer constant or a reference to an integer constant (see [Section 6.5](#)). The value of this field must be a character string of length shorter than the maximum length, where the remaining bytes are padded with the null character.

A `fixed_length_string` used within a [record](#) can be optional at runtime. This is specified by setting the attribute called `optional` to true, else it must be set to false.

The name of the `fixed_length_string` must be specified using its name attribute. The name must be a valid identifier (refer [Appendix A.4](#)) and it must be unique within the scope of the encapsulating record. The interpretation attribute is an optional attribute that may be used to provide a textual description of the `fixed_length_string`.

Examples:

```
<fixed_length_string name="Node_Name" string_length="80" optional="false"/>
```

variable_length_string

A `variable_length_string` is a Simple Field that contains a character string whose maximum length is not fixed. This field is always preceded by a Meta Field called [count_field](#) that specifies its exact length in bytes.

A `variable_length_string` used within a [record](#) can be optional at runtime. This is specified by setting the attribute called `optional` to true, else it must be set to false.

The name of the `variable_length_string` must be specified using its name attribute. The name must be a valid identifier (refer [Appendix A.4](#)) and it must be unique within the scope of the encapsulating record. The interpretation attribute is an optional attribute that may be used to provide a textual description of the `variable_length_string`.

Examples:

```
<variable_length_string name="node_description" interpretation="a brief description of the node's  
functions" optional="true">  
  <count_field field_type_unsigned="unsigned byte"/>  
</variable_length_string>
```

6.2.3 BLOB Fields

BLOB Fields represent large binary data such as JPEG images.

variable_length_field

A `variable_length_field` is a Simple Field that contains a BLOB. This field is always preceded by a Meta Field called [count_field](#) that specifies the size of the BLOB in bytes.

A `variable_length_field` used within a [record](#) can be optional at runtime. This is specified by setting the attribute called `optional` to true, else it must be set to false.

The name of the `variable_length_field` must be specified using its name attribute. The name must be a valid identifier (refer [Appendix A.4](#)) and it must be unique within the scope of the encapsulating record. The interpretation attribute is an optional attribute that may be used to provide a textual description of the `variable_length_field`.

Examples:

```
<variable_length_field name="JPEG_frame" field_format="JPEG" optional="false">  
  <count_field field_type_unsigned="unsigned integer"/>  
</variable_length_field>
```

variable_format_field

A `variable_format_field` is a Simple Field that contains a BLOB whose format can vary at runtime. This field is preceded by an ordered sequence of two Meta Fields called [format_field](#) and [count_field](#) in that order. These Meta Fields specify the format and size of the BLOB in bytes respectively.

A `variable_format_field` used within a [record](#) can be optional at runtime. This is specified by setting the attribute called `optional` to true, else it must be set to false.