

Status: Approved

Version: 1.2

# SunSpec Device Information Model Specification

## SunSpec Specification



### **Abstract**

This document specifies the definition and usage of SunSpec Device Information Models.

Copyright © SunSpec Alliance 2024. All Rights Reserved.

All other copyrights and trademarks are the property of their respective owners.

## **License Agreement and Copyright Notice**

This document and the information contained herein is provided on an "AS IS" basis and the SunSpec Alliance DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY OWNERSHIP RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

This document may be used, copied, and furnished to others, without restrictions of any kind, provided that this document itself may not be modified in any way, except as needed by the SunSpec Technical Committee and as governed by the SunSpec IPR Policy. The complete policy of the SunSpec Alliance can be found at [sunspec.org](http://sunspec.org).

Prepared by the SunSpec Alliance

Website: [sunspec.org](http://sunspec.org)

Email: [info@sunspec.org](mailto:info@sunspec.org)

## Revision History

Version	Date	Comments
1.0	4-20-2021	Initial release
1.1	5-9-2022	Added clarification for read/write timing requirements in section 6.5.
1.2	7-8-2024	Added new fields to the models – detailed description and standards. Mandated the use of function code 6 for single register write (section 6.3). Mandated the support for Broadcast in RTU devices (section 6.3). Set the read/write timing to have a maximum delay of 1000 ms (Section 6.5)

## About the SunSpec Alliance

The SunSpec Alliance is a trade alliance of developers, manufacturers, operators, and service providers together pursuing open information standards for the distributed energy industry. SunSpec standards address most operational aspects of PV, storage, and other distributed energy power plants on the smart grid, including residential, commercial, and utility-scale systems, thus reducing cost, promoting innovation, and accelerating industry growth.

Over 160 organizations are members of the SunSpec Alliance, including global leaders from Asia, Europe, and North America. Membership is open to corporations, non-profits, and individuals. For more information about the SunSpec Alliance, or to download SunSpec specifications at no charge, visit <https://sunspec.org/>.

## About the SunSpec Specification Process

SunSpec Alliance specifications are initiated by SunSpec members to establish an industry standard for mutual benefit. Any SunSpec member can propose a technical work item. Given sufficient interest and time to participate, and barring significant objections, a work group is formed, and its charter is approved by the board of directors. The workgroup meets regularly to advance the agenda of the team.

The output of the workgroup is generally in the form of a SunSpec Interoperability Specification. These documents are considered to be normative, meaning that there is a matter of conformance required to support interoperability. The revision and associated process of managing these documents is tightly controlled. Other documents are informative, or make some recommendations with regard to best practices, but are not a matter of conformance. Informative documents can be revised more freely and more frequently to improve the quality and quantity of information provided.

SunSpec Interoperability Specifications follow a lifecycle pattern of: DRAFT, TEST, APPROVED, and SUPERSEDED.

For more information or to download a SunSpec Alliance specification, go to <https://sunspec.org/specifications/>.

# Table of Contents

1	Introduction .....	9
1.1	Document Organization .....	9
1.2	Terminology .....	10
2	Normative References .....	13
3	Overview.....	14
3.1	Device Information Model Structure.....	14
3.1.1	Model.....	15
3.1.2	Point .....	15
3.1.3	Point Group .....	15
3.1.4	Symbol.....	15
3.1.5	Comment .....	15
3.2	Device Information Model Definition and Instance Relationship.....	15
3.3	Device Information Model Usage .....	16
3.3.1	Modbus .....	18
3.3.2	JSON.....	18
4	Device Information Model Definition .....	19
4.1	Definition Elements .....	19
4.1.1	Model Element.....	19
4.1.2	Point Group Element .....	20
4.1.3	Point Element .....	20
4.1.4	Symbol Element.....	20
4.1.5	Comment Element .....	20
4.2	Element Attributes .....	21
4.2.1	ID.....	22
4.2.2	Points.....	22
4.2.3	Groups .....	22
4.2.4	Type .....	22
4.2.5	Value .....	22
4.2.6	Count.....	24
4.2.7	Size.....	24
4.2.8	Scale Factor.....	24
4.2.9	Units.....	24
4.2.10	Access.....	24
4.2.11	Mandatory .....	24

4.2.12	Label.....	24
4.2.13	Description .....	25
4.2.14	Symbols .....	25
4.2.15	Detailed Description.....	25
4.2.15	Standards.....	25
5	Device Information Model Definition Encoding .....	26
5.1	JSON Message Encoding .....	26
5.1.1	Element Types .....	26
5.1.2	Element Attribute Types .....	26
5.1.3	Model Encoding .....	27
5.1.4	Point Group Encoding.....	28
5.1.5	Point Encoding.....	28
5.1.6	Symbol Encoding .....	29
5.1.7	Comment Encoding .....	29
5.2	CSV Encoding .....	29
5.2.1	Columns.....	29
5.2.2	Rows.....	31
6	Device Information Model Usage for Modbus.....	32
6.1	Device Modbus Map .....	32
6.1.1	Modbus Address Location .....	32
6.1.2	Information Models .....	32
6.1.3	End Model .....	33
6.2	Device Information Model Discovery .....	33
6.3	Modbus Functions .....	33
6.4	Value Representation.....	34
6.4.1	16-bit Integer Values.....	34
6.4.2	32-bit Integer Values.....	34
6.4.3	64-bit Integer Values.....	35
6.4.4	128-bit Integer Values.....	35
6.4.5	String Values .....	35
6.4.6	32-bit Floating-point Values .....	36
6.4.7	64-bit Floating-point Values .....	36
6.4.8	sunssf .....	37
6.5	Verifying Written Values .....	37
6.6	Modbus Error Handling .....	37
6.6.1	Unimplemented Registers .....	38

6.6.2	Writing Invalid Value .....	38
6.6.3	Writing a Read-Only Register .....	38
6.7	Security .....	38
7	Device Information Model Usage for JSON .....	39
	Appendix A: Model Definition Examples.....	41
	Appendix B: Model Instance Examples .....	46

# Index of Tables

- Table 1: Model Definition Elements ..... 19
- Table 2: Element Attributes ..... 21
- Table 3: Point Element Type Attribute Values ..... 23
- Table 4: Point Group Element Type Attribute Values ..... 23
- Table 5: Definition Element JSON Encoding ..... 26
- Table 6: JSON-encoded Element Attribute Types ..... 27
- Table 7: Spreadsheet Column Encoding ..... 30
- Table 8: Modbus 16-bit Integer Value Register ..... 34
- Table 9: Modbus 32-bit Integer Value Registers ..... 34
- Table 10: Modbus 64-bit Integer Value High Registers ..... 35
- Table 11: Modbus 64-bit Integer Value Low Registers ..... 35
- Table 12: Modbus 128-bit Integer Value Registers ..... 35
- Table 13: Modbus String Value Registers ..... 35
- Table 14: Modbus 32-bit Floating-point Value High Register ..... 36
- Table 15: Modbus 32-bit Floating-point Value Low Register ..... 36
- Table 16: Modbus 64-bit Floating-point Register 1 ..... 36
- Table 17: Modbus 64-bit Floating-point Register 2 ..... 36
- Table 18: Modbus 64-bit Floating-point Register 3 ..... 37
- Table 19: Modbus 64-bit Floating-point Value Register 4 ..... 37
- Table 20: Modbus sunssf Value Registers ..... 37
- Table 21: Point Type Mapping to JSON Type ..... 40

# Table of Figures

- Figure 1: Device Information Model Communication and Implementation ..... 9
- Figure 2: Device Information Model Elements ..... 14
- Figure 3: Device Information Model Definition-Instance Map ..... 16
- Figure 4: Device Information Model Instance ..... 17
- Figure 5: Device Modbus Map ..... 32



# 1 Introduction

SunSpec Device Information Models provide a simple, standardized mechanism for specifying data sets supported by a device.

Device Information Models are used to structure device data for exchange across communications interfaces. The following figure shows the communication scenario and the responsibility of the SunSpec device to implement Device Information Models.

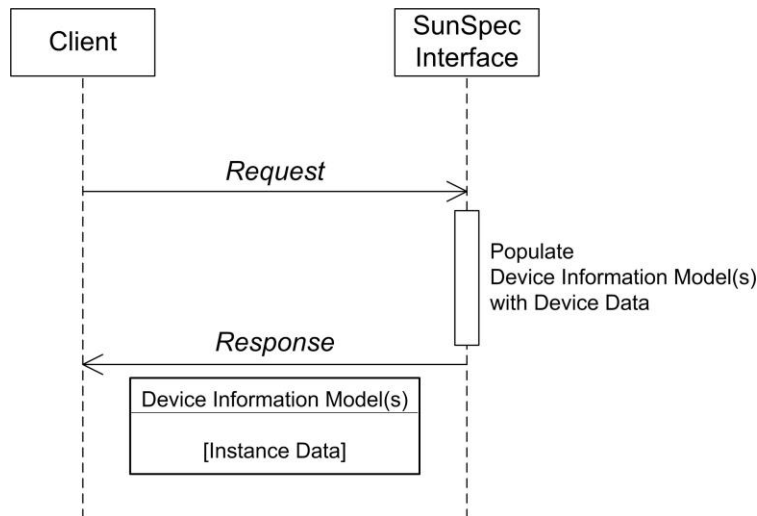


Figure 1: Device Information Model Communication and Implementation

This specification standardizes Device Information Model definition and specifies usage for two information representations:

- Modbus
- JSON encoded messages

## 1.1 Document Organization

Chapter 2 lists the standards documents that are normative references for this document.

Chapter 3 introduces Device Information Model concepts and structure used to define, implement, and use the model.

Chapter 4 provides a formal Device Information Model specification.

Chapter 5 specifies JSON and CSV model definition encoding.

Chapter 6 describes Device Information Model usage for the Modbus messaging structure.

Chapter 7 describes Device Information Model usage for JSON message encoding.

## 1.2 Terminology

Attribute	An attribute describes a definition element or provides additional information about the element. For example, an access attribute is a point element attribute that indicates if a point value is read/write or read-only. Attributes can be required or optional.
CSV	<u>C</u> omma- <u>s</u> eparated <u>V</u> alues are plain text value fields separated by commas. CSV file formats can be opened by spreadsheet program and can be used as a format for data exchange between applications or devices.
Definition element	Definition elements are associated with a Device Information Model and describe the model data structure and usage. A definition element can have a value or provide a container for other elements. The Device Information Model defines the following elements: <ul style="list-style-type: none"><li>• model</li><li>• point</li><li>• point group</li><li>• symbol</li><li>• comment</li></ul> Definition elements have attributes that qualify or describe the element.
Device	A device is an entity that exchanges data across communications interfaces. A device has a data set, modeled by Device Information Models, that describes physical and state information about the device. The device data set is the set of logically-related data points specific to the device type. The collection of Device Information Models that describe the data set corresponds to the full set of device data points supported by the device.

Device Information Model	The Device Information Model is used to structure device data for exchange across communications interfaces. The model provides a mechanism for specifying the data set supported by a device, which consists of a set of standardized definition elements.
Device Information Model definition	A Device Information Model definition specifies the data points that make up the Device Information Model and the usage information associated with each data point. There is one definition for each Device Information Model. Device Information Model definitions represent collections of device data points. The canonical form of Device Information Model definitions is specified using JSON encoding.
Device Information Model instance	A Device Information Model instance is created from a Device Information Model definition. The instance includes data point values specified for each of the defined data points. There can be any number of instances of a Device Information Model.
JSON	<u>J</u> ava <u>S</u> cript <u>O</u> bject <u>N</u> otation is a lightweight format used for data exchange. The canonical form of Device Information Model definitions is specified using JSON encoding. This document specifies JSON encoding for Device Information Model instances.
Modbus	Modbus is a communication protocol for transmitting information between devices using a serial or TCP/IP communication interface. This document specifies Modbus encoding for Device Information Model instances.
Model	A Device Information Model <i>model</i> element defines a logical grouping of <i>points</i> . Each <i>model</i> has a unique model ID.
MUST, MUST NOT, REQUIRED, SHALL, SHALL NOT, SHOULD, SHOULD NOT, RECOMMENDED, MAY, and OPTIONAL	The keywords "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this specification, are to be interpreted as described in IETF RFC 2119.
Point	A Device Information Model <i>point</i> element defines a device data point and has a value.
Point group	A Device Information Model <i>group</i> element contains a group of <i>points</i> and/or other <i>point groups</i> .
Point group, top-level	The top-level point group is the first element of a Device Information Model and contains all other elements.

RESTful web service

A RESTful web service is an architectural style that uses Representational State Transfer (REST) for web applications to access web service resources. REST HTTP methods for access resources include GET, PUT, POST, and DELETE.

Symbol

A Device Information Model *symbol* element defines a name-value pair. It is used to represent a constant value associated with the enumerated value or bit position of a *point*.

UTF-8

UTF-8 is a method for encoding Unicode characters using 8-bit sequences that can include one or more bytes.

## 2 Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", RFC 2279, DOI 10.17487/RFC2279, January 1998, <https://www.rfc-editor.org/info/rfc2279>.

[RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 7159, DOI 10.17487/RFC7159, March 2014, <https://www.rfc-editor.org/info/rfc7159>.

Modbus IDA, MODBUS Application Protocol Specification v1.1b3, North Grafton, Massachusetts, [www.modbus.org/specs.php](http://www.modbus.org/specs.php), April 26, 2012.

Modbus IDA, MODBUS/TCP Security Protocol Specification v21, North Grafton, Massachusetts, [www.modbus.org/specs.php](http://www.modbus.org/specs.php), July 24, 2018.

IEEE 754-2008, IEEE Standard for Floating-Point Arithmetic, August 29, 2008, <https://ieeexplore.ieee.org/servlet/opac?punumber=4610933>.

### 3 Overview

This section introduces Device Information Model definition and usage.

Device Information Model definitions represent collections of device data points. A device implementation based on the information models can use the model definitions to standardize the interface to device data points. This includes logically grouping the information to correspond to the data point grouping requirements of a device.

#### 3.1 Device Information Model Structure

Device Information Models are defined using the following definition elements:

- model
- point
- point grouping
- symbol
- comment

The point definition element represents the Device Information Model data, and the other definition elements govern data structuring and usage. The following figure shows the primary definition elements' structural relationship.

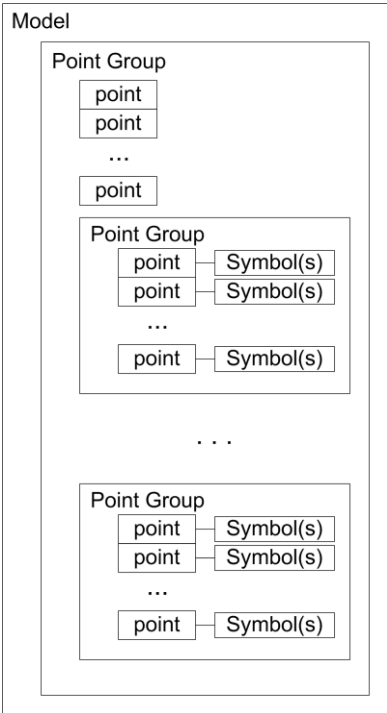


Figure 2: Device Information Model Elements

### 3.1.1 Model

The *model* definition element includes all the definition elements. It is used as the container for the logically related set of device data points, for a particular model.

### 3.1.2 Point

The *point* definition element defines a Device Information Model data point. Point elements hold data values that correspond to a device property. There are typically multiple point definitions in the model definition.

### 3.1.3 Point Group

The *point group* definition element provides a way to logically group a set of points. There are three reasons to group points:

- The top-level model organization construct is always a point group. The first element in a model definition is the top-level point group and includes all the point and point group definitions in the model.
- A repeating set of points can be grouped, creating multiple instances of the point group that can be accessed as an array of point groups.
- A set of points with synchronous operational requirements can be grouped, indicating that the points in the group must be read and written atomically.

### 3.1.4 Symbol

The *symbol* definition element assigns identifiers to values associated with a point definition element. Symbols define the set of valid values for the point and provide identifiers that can be used to represent the value.

### 3.1.5 Comment

The *comment* definition element associates a comment string with any of the other definition elements. It can be used to document the element definition.

## 3.2 Device Information Model Definition and Instance Relationship

It is important to understand the relationship between a Device Information Model *definition* and a Device Information Model *instance*.

A Device Information Model *definition* specifies the data points that make up the Device Information Model and the usage information associated with each data point. There is one definition for each Device Information Model.

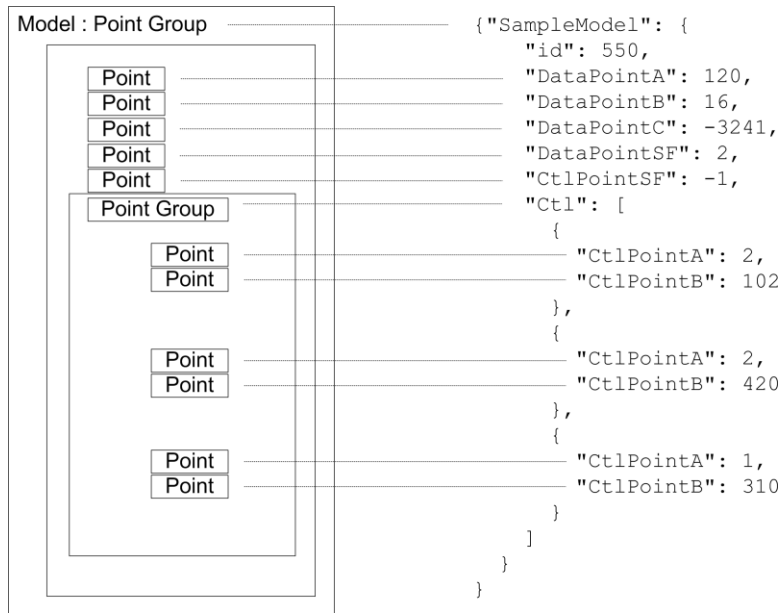


Figure 3: Device Information Model Definition-Instance Map

A Device Information Model *instance* is created from a Device Information Model *definition*. The *instance* includes data point values specified for each of the defined data points. There can be any number of Device Information Model *instances*.

The canonical form of Device Information Model *definitions* is specified in a JSON encoding. For convenience, alternative Device Information Model definition representations can be used provided they preserve all the Device Information Model definition content. In addition to the canonical JSON encoding, this specification standardizes a CSV encoding for Device Information Model definitions to support a spreadsheet presentation of the *definitions*.

### 3.3 Device Information Model Usage

Devices use the Device Information Model definitions that represent the data points supported by the device. Further, a device implementation includes the collection of Device Information Models that correspond to the full set of device data points.



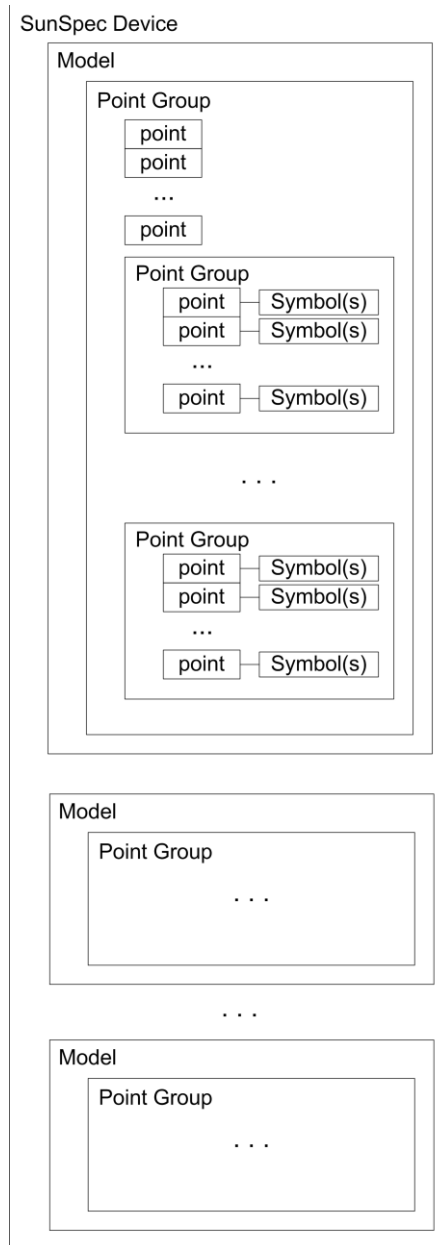


Figure 4: Device Information Model Instance

Information can be exchanged with a device by requesting some or all the device data points, using a communication interface that implements the standardized Device Information Models. Device Information Model definitions are interface-independent. This specification standardizes Device Information Model usage for both Modbus, and JSON data representations.

### **3.3.1 Modbus**

Device Information Models can be mapped into a Modbus address space. A collection of Device Information Models can be used to create a Modbus map that corresponds to the data points supported by the device.

### **3.3.2 JSON**

Device Information Model contents can be represented as JSON objects. Devices can access data points supported by the device through an interface that supports JSON objects, such as a RESTful web service.

## 4 Device Information Model Definition

Device Information Models are defined using standardized elements. Each Information Model Definition includes the definition elements specified in this section.

### 4.1 Definition Elements

A model definition MAY have the following elements:

Element	Description
<i>model</i>	A logical grouping of data points that are assigned a model ID.
<i>group</i>	A group of <i>points</i> or point <i>groups</i> . A <i>model</i> can have multiple point groups and point groups can be nested. A <i>model</i> always has a top-level point group that includes all points and point groups in the model. A <i>model</i> can only have one top-level point group.
<i>point</i>	A data point that has a value.
<i>symbol</i>	A name-value pair used to represent a constant value associated with an enumerated value or bit position in a <i>point</i> .
<i>comment</i>	The text used to annotate the information model definition. Comments are associated with any definition element ( <i>model</i> , <i>group</i> , <i>point</i> , or <i>symbol</i> ) in the <i>model</i> definition.

Table 1: Model Definition Elements

#### 4.1.1 Model Element

The *model* element includes all the other definition elements associated with the model.

A model definition MUST have a top-level point group that includes all the points and point groups in the model.

A model definition MUST include the following two points as the first two element definitions inside the top-level point group:

- model ID with an identifier of  $ID$
- model length with an identifier of  $L$

The value of the  $ID$  point MUST be a SunSpec model ID which is a unique integer between 1 and 65535, inclusive. SunSpec model IDs are administered by the SunSpec Alliance.

The value of the model length  $L$  point MUST be 0 (zero) in the model definition. The model length point may not be used in some encodings. When the model length is used, as in a Modbus map, the length point ( $L$ ) MUST be set to the length of that model instance. Some model definitions have elements that may vary in size, which causes the model instance length to vary.

The order of members of a model element is significant and MUST be maintained.

See Table 2: Element Attributes for the valid mandatory and optional element attributes for a model element definition.

### 4.1.2 Point Group Element

The point *group* element includes point elements or other point group elements.

The point group type MUST either be `group` or `sync`.

The group point group type is used to create a set of points and point groups. If the count for the point group is greater than one, the point group repeats the number of times specified by count. The count attribute can be defined as a constant in the point group definition or be specified as the value of another point in the model definition. If the point group count is specified in another point, that point MUST be defined in the top-level point group of the model before the point group definition.

The sync point group type is used to designate points and point groups that MUST be read and written atomically. Implementations MUST indicate an error if all the members of a sync group are not able to be read or written atomically.

The order of the point group members is significant and MUST be maintained.

All points in a point group MUST be defined before any point groups are defined.

See Table 2: Element Attributes for the valid mandatory and optional element attributes for a point group element definition.

### 4.1.3 Point Element

The *point* element defines a data point element.

The size of the data element MUST be specified for points that have a type of `string`.

If the count for the point is greater than one, the point repeats the number of times specified by the count. The count attribute can be defined as a constant in the point definition or be specified as a value of another point in the model definition. If the point count is specified in another point, that point MUST be defined in the top-level point group of the model before the point definition.

See Table 2: Element Attributes for the valid mandatory and optional element attributes for a point element definition.

### 4.1.4 Symbol Element

The *symbol* element associates an ID with a constant point value in a point definition.

The symbol element MUST be associated with a point definition. A point definition MAY have multiple symbols associated with it. Each symbol ID MUST be unique for that point definition. The symbol definitions for a point definition serve as a set of possible enumerated values that are valid for the point. If a point has associated symbols defined, all values not in the set of symbol definitions MUST be considered invalid by an implementation.

See Table 2: Element Attributes for the valid mandatory and optional element attributes for a symbol element definition.

### 4.1.5 Comment Element

The *comment* element is a single string and is associated with any valid definition element other than a *comment* element. A definition element MAY have multiple comments associated with it.

The *comment* element permits additional element definition annotation beyond the element definition attributes shown in Table 2: Element Attributes.

## 4.2 Element Attributes

Definition elements include attributes that qualify or describe the element. Table 2: Element Attributes shows the attributes associated with each element definition type. The table also uses the following notation to indicate which attributes are associated with each element type:

- Model
- Point Group
- Point
- Symbol

Additionally, for each element type, **R** indicates the attribute is required in an element definition and **O** indicates the attribute is optional.

Attribute	Description	M	G	P	S
<i>ID</i>	The element ID.	R	R	R	R
<i>Points</i>	An array of point definitions in a point group.		R		
<i>Group</i>	An array of point elements or other point group elements.	R			
<i>Groups</i>	An array of point group definitions in a point group.		O		
<i>Value</i>	If present, a constant value associated with the element.			O	R
<i>Type</i>	The element type.		R	R	
<i>Count</i>	The occurrence count of the element.		O	O	
<i>Size</i>	The element size. Mandatory when type is <code>string</code> .			O	
<i>Scale Factor</i>	If present, the scale factor point associated with the element.			O	
<i>Units</i>	If present, the units associated with the element.			O	
<i>Access</i>	Element access, read or read/write. If not present, defaults to read. (R or RW)			O	
<i>Mandatory</i>	Element is mandatory/optional. If not present, default to optional. (M or O)			O	
<i>Label</i>	Short label associated with the element.	R	R	O	O
<i>Description</i>	Description associated with the element.	O	O	O	O
<i>Symbols</i>	A name-value pair used to represent a constant value associated with an enumerated value or bit position in a point.	O	O	O	O
<i>Detailed Description</i>	A detailed note to describe the usage of the point. This attribute may include examples.	O	O	O	O
<i>Standards</i>	This list mentions all the standards/grid codes in which the point is mandatory for compliance.	O	O	O	O

Table 2: Element Attributes

Optional attributes may have a default value and the default value may be different for different

element types.

If an attribute does not have an entry in the table for an element type, the attribute **MUST NOT** be used in an element definition for that element type.

#### **4.2.1 ID**

The *ID* attribute is the element name and **MUST** be unique in the immediate group in which it is defined. An ID **MUST** consist of only alphanumeric characters and the underscore character. The *ID* attribute for a model element **MUST** be the numeric SunSpec model id.

#### **4.2.2 Points**

The *points* attribute is a point definition array of points contained in the point group.

#### **4.2.3 Groups**

The *groups* attribute is a point group definitions array of point groups contained in the point group.

#### **4.2.4 Type**

The *type* attribute is the element type. Table 3: Point Element Type Attribute Values describes the possible type values for point elements and Table 4: Point Group Element Type Attribute Values specifies the possible type value for group elements.

#### **4.2.5 Value**

The *value* attribute is the constant value associated with the element. If the element does not have a constant value, the value attribute **MUST** be omitted.

If a point does not have a valid value, the unimplemented value **MUST** be used for the value. During device operation, the point value **MAY** change from the unimplemented value to a valid value or from a valid value to the unimplemented value at any time.

If the element contains a constant value, its model definition file encoding **SHALL** follow the conventions listed in Table 16: Point Type Mapping to JSON Type.

Type	Description
int16	Signed 16-bit integer
int32	Signed 32-bit integer
int64	Signed 64-bit integer
raw16	16-bit raw value
uint16	Unsigned 16-bit integer
uint32	Unsigned 32-bit integer
uint64	Unsigned 64-bit integer
acc16	Unsigned 16-bit accumulator (deprecated in favor of uint16)
acc32	Unsigned 32-bit accumulator (deprecated in favor of uint32)
acc64	Unsigned 64-bit accumulator (deprecated in favor of uint64)
bitfield16	16-bit bitfield
bitfield32	32-bit bitfield
bitfield64	64-bit bitfield
enum16	16-bit enumeration
enum32	32-bit enumeration
float32	32-bit floating-point
float64	64-bit floating-point
string	String (UTF-8 encoded)
sunssf	Scale factor – Signed power of 10 multiplier (+) or divider (-)
pad	16-bit pad used for alignment
ipaddr	IP Address as an unsigned 32-bit.
ipv6addr	16-byte IP V6 address
eui48	48-bit MAC address

Table 3: Point Element Type Attribute Values

Type	Description
group	Group
sync	Synchronization group

Table 4: Point Group Element Type Attribute Values

#### 4.2.6 Count

The *count* attribute specifies the number of occurrences of the element in the model. The count is commonly used to specify the number of occurrences of a point group but it may also be used to specify a single repeating point.

The count MAY be specified as a constant value in the model definition, or by another point in the model that contains the count.

If the count is specified by another point in the model, the specifying point MUST be defined in the top-level point group before the element that the count applies to. The value of a point containing a count MUST be static and not change over time.

#### 4.2.7 Size

The *size* attribute specifies the maximum element length in 16-bit words. The *size* attribute MUST be provided for the string point type and MAY be provided for the pad type. The *size* attribute MUST not be provided for any other type.

#### 4.2.8 Scale Factor

As an alternative to floating-point format, values are represented by integer values with a signed scale factor applied. A negative scale factor explicitly shifts the decimal point to the left, and a positive scale factor shifts the decimal point to the right by the number of places specified in the scale factor value.

The *scale factor* attribute specifies a scale factor to be used with the point element. The scale factor may be another point defined in the model or a constant value. If the scale factor specifies another point defined in the model, the referenced point MUST be defined as a scale factor type (`sunssf`).

If a constant value is specified, the value MUST be a valid scale factor multiplier.

The value of a scale factor point MUST be static and MUST NOT change over time.

#### 4.2.9 Units

The *units* attribute is a string that specifies the units associated with the element.

Units are defined as needed by specific models. Where units are shared across models, care is taken to ensure a common definition of those units.

#### 4.2.10 Access

The *access* attribute specifies if the element is writable or read-only. If specified, the value MUST be read-only (R) or read/write (RW). If not specified, the default access is read-only.

#### 4.2.11 Mandatory

The *mandatory* attribute specifies whether the element is required to be implemented. If specified, the value MUST be either mandatory (M) or optional (O). If not specified, the default value is optional. Points specified as mandatory MUST always have a valid value. Points specified as optional may have the unimplemented value for the corresponding point type.

#### 4.2.12 Label

The *label* attribute specifies a short label associated with the element.



#### **4.2.13 Description**

The *description* attribute provides a brief description of the element..

#### **4.2.14 Symbols**

A name-value pair used to represent a constant value associated with an enumerated value or bit position in a point.

#### **4.2.15 Detailed Description**

The detailed description of a point adds information to clarify the usage or special requirements for the point. This may include examples to clearly define the usage of the point.

#### **4.2.15 Standards**

This field is a list of all the standards for which the point is required/mandatory. The Standards field helps in mapping the requirements of a specific grid code/standard.

## 5 Device Information Model Definition Encoding

The canonical format used to define SunSpec Device Information Models is JSON.

An alternative CSV encoding is also specified in this document to support a spreadsheet presentation of Device Information Model definitions.

### 5.1 JSON Message Encoding

This section describes the method of representing Device Information Model definitions in JSON.

Model definitions defined in JSON MUST be encoded using UTF-8.

#### 5.1.1 Element Types

Table 5: Definition Element JSON Encoding shows the JSON name and value type used for element type definitions.

Element	JSON Name	JSON Value
Model	<code>model</code>	Object of model elements
Point Group	<code>groups</code>	Object of group elements
Point	<code>points</code>	Object of point attributes
Symbol	<code>symbols</code>	Array of symbol objects
Comment	<code>comments</code>	Array of comment strings

Table 5: Definition Element JSON Encoding

#### 5.1.2 Element Attribute Types

Table 6: JSON-encoded Element Attribute Types shows the JSON name and value type used for element attribute type definitions.

Attribute	JSON Name	JSON Values
<i>ID</i>	id	
<i>Value</i>	value	
<i>Type</i>	type	int16, int32, int64, uint16, raw16, uint32, uint64, acc16, acc32, acc64, bitfield16, bitfield32, bitfield64, enum16, enum32, float32, float64, string, sunssf, pad, ipaddr, ipv6addr, eui48, group, sync
<i>Count</i>	count	
<i>Size</i>	size	
<i>Scale Factor</i>	sf	
<i>Units</i>	units	
<i>Access (R/RW)</i>	access	R, RW
<i>Mandatory (M/O)</i>	mandatory	M, O
<i>Label</i>	label	
<i>Description</i>	desc	
<i>Detailed Description</i>	detail	
<i>Standards</i>	standards	Array of standards strings, e.g., [IEEE 1547-2018]

Table 6: JSON-encoded Element Attribute Types

### 5.1.3 Model Encoding

A model definition **MUST** be represented as an object with a single property named `model` and an object as the value. See Appendix A for model definition examples.

The object value of `model` **MUST** have two properties: `id` and `group`. The value of the `id` property **MUST** be the SunSpec numeric model ID.

The value of the `group` property **MUST** be an object that includes the contents of the rest of the model definition. The `group` property represents the required single top-level point group in the model.

The `model` object **MUST** have a `label` property and **MAY** have `desc`, and `comments` properties.

The following example shows the model element encoding.

```
{
  "model": {
    "id": <model id>,
    "group": {
      <rest of model content>
    }
    "label": <model label>,
    "desc": <model description>
  }
}
```

#### 5.1.4 Point Group Encoding

A point group definition **MUST** be represented as an object with the required and optional properties for a point group.

A point group definition **MUST** have a property named `type` that has a group value identifying the object as a point group definition.

A point group definition **MAY** have a property named `comments` that is an array holding the comment strings associated with the point group.

The following example shows the point group element encoding.

```
{
  "name": <point group name>,
  "type": <point group type>,
  "count": <point group count>,
  "label": <point group label>,
  "desc": <point group description>,
  "points": [<points>],
  "groups": [<point groups>], "comments": [<comment strings>]}
```

#### 5.1.5 Point Encoding

A point definition **MUST** be represented as an object with the required and optional point properties for a point.

A point definition **MUST** have a property named `type` that has a value of `point` identifying the object as a point definition.

A point definition **MAY** have a property named `symbols` that is an array holding the symbols associated with the point.

A point definition **MAY** have a property named `comments` that is an array holding the comment strings associated with the point.

The following example shows the point element encoding.

```
{
  "name": <point name>,
  "value": <point value>,
  "type": <point type>,
  "count": <point count>,
  "size": <point size>,
  "sf": <point scale factor>,
  "units": <point units>,
  "access": <point access>,
  "mandatory": <point mandatory>,
  "label": <point label>,
  "desc": <point description>,
  "symbols": [<symbols>],
  "comments": [<comment strings>],
  "detail": <point detailed description>,
  "standards": [<standards>]
}
```

### 5.1.6 Symbol Encoding

A symbol definition **MUST** be represented as an object with the required and optional properties for a symbol.

A symbol definition **MAY** have a property named `comments` that is an array holding the comment strings associated with the symbol.

The following example shows the symbol element encoding.

```
{
  "name": <symbol name>,
  "value": <point value>,
  "label": <point label>,
  "desc": <point description>,
  "comments": [<comment strings>]
}
```

### 5.1.7 Comment Encoding

A comment definition **MUST** be represented as a string. Comments are associated with other elements as an array of comments in the element definition.

## 5.2 CSV Encoding

There is a one-to-one mapping between CSV model definition attributes and the JSON definition. The JSON encoding is the canonical form of a Device Information Model, and the CSV encoding is supported for convenience in creating and inspecting model definitions using a spreadsheet application.

A spreadsheet renders the encoding as a row and column matrix. Each row in the spreadsheet defines a model definition element. Each column represents an element attribute. The CSV encoding could be instantiated in several ways, such as an Excel spreadsheet.

The CSV encoding is defined such that a JSON encoding can be generated from the CSV encoding.

### 5.2.1 Columns

The column names in Table 7: Spreadsheet Column Encoding specified as mandatory **MUST** be used as the column names in the spreadsheet encoding. Other columns **MAY** be included in the encoding at any column location. The names specified as optional in the table are included for convenience.

Column Name	Mandatory/Optional
<b>Name</b>	Mandatory
<b>Value</b>	Mandatory
<b>Type</b>	Mandatory
<b>Count</b>	Mandatory
<b>Size</b>	Mandatory
<b>Scale Factor</b>	Mandatory
<b>Units</b>	Mandatory
<b>RW Access (RW)</b>	Mandatory
<b>Mandatory (M)</b>	Mandatory
<b>Static (S)</b>	Mandatory
<b>Label</b>	Mandatory
<b>Description</b>	Mandatory
<b>Detailed Description</b>	Mandatory
<b>Standards</b>	Mandatory
<b>Notes</b>	Optional (used for draft models, not included in approved models)
<b>Address Offset</b>	Optional. Offset of the element from the beginning of the Information Model, if known. Typically generated from definition information.
<b>Group Offset</b>	Optional. Offset of the element from the beginning of the immediate containing group. Typically generated from definition information.

Table 7: Spreadsheet Column Encoding

The **Name**, **Value**, and **Type** fields are used to determine the definition element type. The following rules specify how to interpret each element when a spreadsheet is used to define a model.

- Model**            **Model** is the model ID as represented in the value of the point **ID**.
- Point group**    **Type** is a point group type. Because point groups can be nested, point group IDs reflect the hierarchy of the point groups. A dot ( . ) delimits hierarchical levels.
- Point**            **Type** is a point type.
- Symbol**            The symbol is a key-value pair associated with the last point defined.
- Comment**        Any line with no **Type** and no **Value** values. The comment only consists of the contents of the first column value in the row. A comment is associated with the next defined element. An element may have more than one comment associated with it.
- Blank line**        Any row with no value in any column. Blank lines are not preserved in the model definition.

### 5.2.2 Rows

Rows in the spreadsheet **MUST** consist of all the point group, point, symbol, and comment elements included in the model definition. The sequence of ordered elements **MUST** be preserved.

The name strings specified in the JSON encoding **MUST** be used for all type and type value representations.

To represent the point group hierarchy, the ID of any point or point group not defined in the top-level point group **MUST** have each point group to which the element belongs below the top-level point group prepend to its ID using a period (.) as a separator between IDs.

## 6 Device Information Model Usage for Modbus

This section specifies Modbus Device Information Model instance encoding. An Information Model instance includes the values associated with the defined content of a model.

### 6.1 Device Modbus Map

Device Information Models are used to construct a device Modbus map. The Information Models that represent the functionality implemented in the device are placed contiguously in the Modbus address space at a defined location as specified in this section.

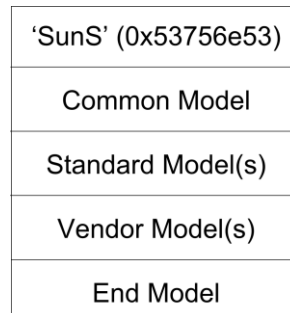


Figure 5: Device Modbus Map

All SunSpec Device Information Model maps **MUST** begin with the `SunS` identifier. The identifier is followed sequentially by common, standard, and vendor Device Information Models, as needed. An end model terminates the map. There is no requirement for any model ordering other than the end model must be the last model in the sequence.

#### 6.1.1 Modbus Address Location

All Modbus device maps **MUST** be in the holding register address space.

The beginning of the device Modbus map **MUST** be located at one of three Modbus addresses in the Modbus holding register address space: 0, 40000 (0x9C40) or 50000 (0xC350). These Modbus addresses are the full 16-bit, 0-based addresses in the Modbus protocol messages.

The first two Modbus registers at the start address **MUST** have the following well-known constant values as a marker: 0x5375, 0x6E53 (hexadecimal values of the ASCII string `SunS`).

#### 6.1.2 Information Models

The Device Information Models **MUST** be placed contiguously, beginning immediately after the `SunS` marker registers. Each Information Model **MUST** have registers corresponding to all the points in the Information Model, including those specified as optional or unimplemented. Points in a model **MUST** be placed such that there are no additional Modbus registers between points specified in the model definition. Points that are not supported or have no valid value **MUST** be assigned the appropriate unimplemented value based on the point type. There **MUST NOT** be additional Modbus registers between Information Models in the device Modbus map.

The length point ( $\mathbb{L}$ ) in an information model instance **MUST** be set to the remaining number of Modbus registers in the model following the length point.



A Modbus register map representing a Device Information Model MUST preserve the order of points and groups defined within the model. Such Modbus register maps MUST have group point elements placed before group elements, at lower register indices, and all instances of repeated point or group elements MUST be consecutive.

### 6.1.3 End Model

The last Information Model in the device Modbus map MUST be a two-register empty model with a model ID of 0xFFFF and a model length of 0.

## 6.2 Device Information Model Discovery

A discovery mechanism can be employed to determine the type and location of each of the Information Models in the device map.

Device architects may choose to implement different collections of Information Models in arbitrary order. In any implementation, after the Modbus address of a particular model is determined, the Modbus location of the points in the model are then known based on the model definition.

All Information Models start with an id register and a length register. This information is used to step through or scan the Information Models even if the ID and contents of an Information Model are not understood by the scanning application. This permits implementations to find and use the Device Information Model(s) they understand and ignore those whose definitions are unknown.

The following procedure is used for Information Model discovery:

1. Read the contents of addresses 0, 40000, and 50000 until the well-known marker is found.
2. Repeat the following steps until a model id of 0xFFFF is found:
  - 1) Read the next two registers to get the ID and length of the next Information Model.
  - 2) Add the length to the Modbus address of the next register after the length register to determine the starting address of the subsequent Information Model.
3. When this process is complete, the Modbus address and ID of each Information Model is known.

## 6.3 Modbus Functions

The Modbus interface MUST comply with the Modbus standard for the functionality specified in this section.

The interface MUST support function code 3 (Read Holding Registers), function code 6 (Write Single Register), and function code 16 (0x10) (Write Multiple Registers).

If Modbus support is provided in the device, it MUST support a Modbus serial interface and/or a Modbus TCP/IP interface.

In addition, Modbus RTU devices MUST support “broadcast mode” as defined in [MODBUS over Serial Line Specification and Implementation Guide V1.02](#).

## 6.4 Value Representation

Values are stored in big-endian order and MUST be compliant with the Modbus specification. All integer values are documented as signed or unsigned. All signed values are represented using a two's-compliment format.

### 6.4.1 16-bit Integer Values

16-bit integers are stored using one register in big-endian order.

Modbus Register	1															
Byte	0								1							
Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Table 8: Modbus 16-bit Integer Value Register

int16 Range: -32767... 32767	NOT IMPLEMENTED value: 0x8000
uint16 Range: 0 ... 65534	NOT IMPLEMENTED value: 0xFFFF
raw16 Range:	
acc16 Range: 0 ... 65535	NOT ACCUMULATED value: 0x0000
enum16 Range: 0 ... 65534	NOT IMPLEMENTED value: 0xFFFF
bitfield16 Range: 0 ... 0x7FFF	NOT IMPLEMENTED value: 0xFFFF
Pad Range: 0x8000	Always returns 0x8000

### 6.4.2 32-bit Integer Values

32-bit integers are stored using two registers in big-endian order.

Modbus Register	1				2			
Byte	0		1		2		3	
Bits	31 ... 24		23 ... 16		15 ... 8		7 ... 0	

Table 9: Modbus 32-bit Integer Value Registers

int32 Range: -2147483647 ... 2147483647	NOT IMPLEMENTED value: 0x80000000
uint32 Range: 0 ... 4294967294	NOT IMPLEMENTED value: 0xFFFFFFFF
acc32 Range: 0 ... 4294967295	NOT ACCUMULATED value: 0x00000000
enum32 Range: 0 ... 4294967294	NOT IMPLEMENTED value: 0xFFFFFFFF
bitfield32 Range: 0 ... 0xFFFFFFFF	NOT IMPLEMENTED value: 0xFFFFFFFF
ipaddr 32 bit IPv4 address	NOT CONFIGURED value: 0x00000000

### 6.4.3 64-bit Integer Values

64-bit integers are stored using four registers in big-endian order.

Modbus Register	1		2	
Byte	0	1	2	3
Bits	63 ... 56	55 ... 48	47 ... 40	39 ... 32

Table 10: Modbus 64-bit Integer Value High Registers

Modbus Register	3		4	
Byte	4	5	6	7
Bits	31 ... 24	23 ... 16	15 ... 8	7 ... 0

Table 11: Modbus 64-bit Integer Value Low Registers

int64 Range: -9223372036854775807 ...  
9223372036854775807

NOT IMPLEMENTED value:  
0x8000000000000000

uint64 Range: 0 ... 18,446,744,073,709,551,615

NOT IMPLEMENTED value:  
0xFFFFFFFFFFFFFFFF

acc64 Range: 0 ... 9223372036854775807

NOT ACCUMULATED value: 0

### 6.4.4 128-bit Integer Values

128-bit integers are stored using eight registers in big-endian order.

ipv6addr 128 bit IPv6 address	Not Configured: 0
-------------------------------	-------------------

Table 12: Modbus 128-bit Integer Value Registers

ipv6addr 128 bit IPv6 address

NOT CONFIGURED value: 0

### 6.4.5 String Values

Store variable length string values in a fixed size register range using a NULL (0 value) to terminate or pad the string. For example, up to 16 characters can be stored in 8 contiguous registers as follows.

Modbus Register	1		2		3		4		5		6		7		8	
Byte	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Character	E	X	A	M	P	L	E	spc	S	T	R	I	N	G	!	NULL

Table 13: Modbus String Value Registers

NOT\_IMPLEMENTED value: all registers filled with NULL, or 0x0000

It is recommended that an empty string be represented with the first register, with a value of 0x0080.

Strings MUST be UTF-8 encoded.

### 6.4.6 32-bit Floating-point Values

Floating-point 32-bit values are encoded according to the IEEE 754 floating-point standard.

Modbus Register	1															
Byte	0								1							
Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
IEEE 754	sign	Exponent							Fraction							

Table 14: Modbus 32-bit Floating-point Value High Register

Modbus Register	2															
Byte	2								3							
Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IEEE 754	Fraction least															

Table 15: Modbus 32-bit Floating-point Value Low Register

float32 Range: see IEEE 754

NOT\_IMPLEMENTED value: 0x7FC00000 (NaN)

### 6.4.7 64-bit Floating-point Values

Floating-point 64-bit values are encoded according to the IEEE 754 floating-point standard.

Modbus Register	1																
Byte	0								1								
Bits	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	
IEEE 754	sign	Exponent							Exponent				Fraction				

Table 16: Modbus 64-bit Floating-point Register 1

Modbus Register	2															
Byte	2								3							
Bits	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
IEEE 754	Fraction															

Table 17: Modbus 64-bit Floating-point Register 2

Modbus Register	3															
Byte	4								5							
Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
IEEE 754	Fraction															

Table 18: Modbus 64-bit Floating-point Register 3

Modbus Register	4															
Byte	6								7							
Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IEEE 754	Fraction															

Table 19: Modbus 64-bit Floating-point Value Register 4

float64 Range: see IEEE 754

NOT IMPLEMENTED value:  
0x7FF8000000000000

### 6.4.8 sunssf

SunSpec scale factor.

Modbus Register	1															
Byte	0								1							
Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Table 20: Modbus sunssf Value Registers

sunssf signed range: -10 ... 10

NOT IMPLEMENTED value: 0x8000

## 6.5 Verifying Written Values

It is common practice with Modbus devices to perform a read operation after a write operation to verify that the values were successfully written. The written values should be read with a maximum delay of 1000 ms for a successful write operation.

When a read operation is performed, the values returned MUST reflect the values contained in the last write request that returned a successful response. The successful write and subsequent read of the values only confirms successful information exchange and does not imply any updates to operational behavior of the device associated with the information.

## 6.6 Modbus Error Handling

This section describes the required Modbus error handling procedures.

### **6.6.1 Unimplemented Registers**

When reading an unimplemented register, the unimplemented value for the data point type MUST be returned.

When writing an unimplemented register, a Modbus exception MUST be generated. The Modbus exception MUST be either exception code 2, 3, or 4.

### **6.6.2 Writing Invalid Value**

When writing an invalid value to a register, a Modbus exception MUST be generated. The Modbus exception MUST be either exception code 2, 3, or 4.

If the register definition specifies a set of enumerated values and not all values are supported, the invalid value behavior applies. If the register only supports a single valid value, the register MUST remain writable and accept the writing of the valid value. The register MUST NOT be implemented as read-only.

### **6.6.3 Writing a Read-Only Register**

When writing a read-only register, a Modbus exception MUST be generated. The Modbus exception MUST be either exception code 2, 3, or 4.

## **6.7 Security**

Modbus/TCP security SHALL be compliant with the Modbus/TCP Security specification [https://modbus.org/docs/MB-TCP-Security-v21\\_2018-07-24.pdf](https://modbus.org/docs/MB-TCP-Security-v21_2018-07-24.pdf).

## 7 Device Information Model Usage for JSON

This section specifies JSON Information Model instance encoding. An Information Model instance contains the values associated with the defined content of a model.

The following rules apply for mapping Information Model content to a JSON encoded object:

- A Model element is represented as a JSON object.
- A Point Group element is represented as a JSON object.
- A Point element is represented as a JSON number or a JSON string. The mapping of each point type is shown in Table 21: Point Type Mapping to JSON Type.
- Repeating elements are represented as a JSON array containing the repeated elements.
- Unimplemented values are omitted or contain a value of null.

Type	JSON Encoding
int16	number
int32	number
int64	number
raw16	number
uint16	number
uint32	number
uint64	number
acc16	number
acc32	number
acc64	number
bitfield16	number
bitfield32	number
bitfield64	number
enum16	number
enum32	number
float32	number
float64	number
string	string
sf	number
pad	N/A
ipaddr	number or hexadecimal string or convert to IP address string (x.x.x.x)
ipv6addr	hexadecimal string or convert to IP address string
eui48	number or hexadecimal string or convert to MAC string (xx:xx:xx:xx:xx:xx)

Table 21: Point Type Mapping to JSON Type



# Appendix A: Model Definition Examples

This appendix contains examples of model definitions using both CSV encoding and canonical JSON encoding. The same, simple sample model definition is used throughout the document for both model definition and instance examples.

The example model definition is a simple model containing three points and a point group that contains two points. The point group count is contained as a data point. The model is 550 and the name of the model point group is *SampleModel*.

## Spreadsheet Model Definition Example

The spreadsheet encoding provides a convenient mechanism for easily viewing the contents of a device information model. The basis of the spreadsheet visualization is CSV encoding. This example shows the CVS representation of the sample model definition and an example of the information in spreadsheet form:

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
	Address Offset	Group Offset	Name	Value	Count	Type	Size	Scale Factor	Units	RW Access (RW)	Mandatory (M)	Static (S)	Label	Description
1														
2	Sample model definition to illustrate different model definition elements													
3			SampleModel			group							Sample Model Label	Sample model model.
4	0		ID	550		uint16					M		Sample Model name	Sample model model name.
5	1		L	0		uint16					M		Sample Model Length	Sample model model length.
6	2		DataPointA			int32		DataPointSF					Data Point A	Data point A description.
7	4		DataPointB			uint16							Data Point B	Data point B description.
8	5		DataPointC			int16							Data Point C	Data point C description.
9	6		DataPointSF			sunssf							Data Point Scale Factor	
10	7		CtlPointSF			sunssf							Control Point Scale Factor	
11	8		CtlCount			uint16							Count Point Group Count	
12	9		Pad			pad								
13	Control point group containing the control elements that repeat													
14			SampleModel.Ctl		CtlCount	group							Control Points	Control point group description.
15	0		CtlPointA			enum16				RW			Control Point A	Control point A description.
16	Control Point A enumerated values													
17			VALUE_A	1									Value A	Control point A value A description.
18			VALUE_B	2									Value B	Control point A value B description.
19			VALUE_C	3									Value C	Control point A value C description.
20	1		CtlPointB			int16		CtlPointSF		RW			Control Point B	Control point B description.

In the example, visual cues, such as color-coding point groups, are added for clarity. A spreadsheet representation should not add to the model definition contents but may use visual elements to help understand the model definition contents.

## CSV Model Definition Encoding Example

The CSV encoding is the basis of the spreadsheet representation. As specified in the CSV encoding section, the group hierarchy is represented by including the group hierarchy in the group name. In this example, the Ctl group has the name of SampleModel.Ctl to explicitly indicate the group hierarchy.

Address Offset,Group Offset,Name,Value,Count,Type,Size,Scale Factor,Units,RW  
Access (RW),Mandatory (M),Static (S),Label,Description, Detailed Description,  
Standards

Sample model definition to illustrate different model definition  
elements,,,,,,,,,,,,,

,,SampleModel,,,group,,,,,,,,,Sample Model Label,Sample model model.

0,,,ID,550,,,uint16,,,,,M,,Sample Model name,Sample model model name,ID must be  
unique,Standard1.

1,,,L,0,,,uint16,,,,,M,,Sample Model Length,Sample model model length,,Standard1.

2,,,DataPointA,,,int32,,DataPointSF,,,,,Data Point A,Data point A  
description,,Standard1.

4,,,DataPointB,,,uint16,,,,,Data Point B,Data point B description,,Standard1.

5,,,DataPointC,,,int16,,,,,Data Point C,Data point C description,,.

6,,,DataPointSF,,,sunssf,,,,,Data Point Scale Factor,,,Standard1,

7,,,CtlPointSF,,,sunssf,,,,,Control Point Scale Factor,

8,,,CtlCount,,,uint16,,,,,Count Point Group Count,

9,,,Pad,,,pad,,,,,,,,,

Control point group containing the control elements that repeat,,,,,,,,,,,,,

,,SampleModel.Ctl,,CtlCount,group,,,,,,,,,Control Points,Control point group  
description,,.

,0,CtlPointA,,,enum16,,,,,RW,,,Control Point A,Control point A description,,.

Control Point A enumerated values,,,,,,,,,,,,,

,,VALUE\_A,1,,,,,,,,,Value A,Control point A value A description,,.

,,VALUE\_B,2,,,,,,,,,Value B,Control point A value B description,,.

,,VALUE\_C,3,,,,,,,,,Value C,Control point A value C description,,.

,1,CtlPointB,,,int16,,CtlPointSF,,RW,,,Control Point B,Control point B  
description,,.

## JSON Model Definition Encoding Example

This example shows the canonical JSON encoding of the device information model definition.

```
{
  "group": {
    "comments": [
      "Sample model definition to illustrate different model definition elements"
    ],
    "desc": "Sample model model.",
    "groups": [
      {
        "comments": [
          "Control point group containing the control elements that repeat"
        ],
        "count": "CtlCount",
        "desc": "Control point group description.",
        "label": "Control Points",
        "name": "Ctl",
        "points": [
          {
            "access": "RW",
            "desc": "Control point A description.",
            "label": "Control Point A",
            "name": "CtlPointA",
            "size": 1,
            "symbols": [
              {
                "comments": [
                  "Control Point A enumerated values"
                ],
                "desc": "Control point A value A description.",
                "label": "Value A",
                "name": "VALUE_A",
                "value": 1
              },
              {
                "desc": "Control point A value B description.",
                "label": "Value B",
                "name": "VALUE_B",
                "value": 2
              },
              {
                "desc": "Control point A value C description.",
                "label": "Value C",
                "name": "VALUE_C",
                "value": 3
              }
            ]
          },
          {
            "type": "enum16"
          }
        ],
        "access": "RW",
        "desc": "Control point B description.",
        "label": "Control Point B",
        "name": "CtlPointB",
        "sf": "CtlPointSF",
        "size": 1,
        "type": "int16"
      }
    ],
    "type": "group"
  }
},
1,
```

```

"label": "Sample Model Label",
"name": "SampleModel",
"points": [
  {
    "desc": "Sample model model name.",
    "label": "Sample Model name",
    "mandatory": "M",
    "name": "ID",
    "size": 1,
    "type": "uint16",
    "value": 550,
    "detail": "ID must be unique",
    "standards": ['Standard1']
  },
  {
    "desc": "Sample model model length.",
    "label": "Sample Model Length",
    "mandatory": "M",
    "name": "L",
    "size": 1,
    "type": "uint16",
    "value": 0,
    "standards": ['Standard1']
  },
  {
    "desc": "Data point A description.",
    "label": "Data Point A",
    "name": "DataPointA",
    "sf": "DataPointSF",
    "size": 2,
    "type": "int32",
    "standards": ['Standard1']
  },
  {
    "desc": "Data point B description.",
    "label": "Data Point B",
    "name": "DataPointB",
    "size": 1,
    "type": "uint16",
    "standards": ['Standard1']
  },
  {
    "desc": "Data point C description.",
    "label": "Data Point C",
    "name": "DataPointC",
    "size": 1,
    "type": "int16",
    "standards": []
  },
  {
    "label": "Data Point Scale Factor",
    "name": "DataPointSF",
    "size": 1,
    "type": "sunssf"
    "standards": ['Standard1']
  },
  {
    "label": "Control Point Scale Factor",
    "name": "CtlPointSF",
    "size": 1,
    "type": "sunssf",
    "standards": []
  },
  {

```

```
        "label": "Count Point Group Count",
        "name": "CtlCount",
        "size": 1,
        "type": "uint16",
        "standards": []
    },
    {
        "name": "Pad",
        "size": 1, "type":
        "pad"
    }
],
    "type": "group"    },
    "id": 550
}
```

## Appendix B: Model Instance Examples

This appendix contains examples of a model instance based on the sample model definition shown in Appendix A. The model instance examples show the contents of the model definition with the current values associated with each data point.

### JSON Message Encoding Example

This example shows a JSON encoding of the sample model with associated values.

```
{"SampleModel": {
  "id": 550,
  "DataPointA": 120,
  "DataPointB": 16,
  "DataPointC": -3241,
  "DataPointSF": 2,
  "CtlPointSF": -1,
  "CtlCount": 3,
  "Ctl": [
    {
      "CtlPointA": 2,
      "CtlPointB": 102
    },
    {
      "CtlPointA": 2,
      "CtlPointB": 420
    },
    {
      "CtlPointA": 1,
      "CtlPointB": 310
    }
  ]
}
```

### MODBUS Message Encoding Example

The example shows a simplified Modbus map that contains the initial marker, sample model, and end model. This shows the general Modbus map structure but is not complete because an actual Modbus map contains additional models.

Modbus Address	Register Contents	Description
40000	'Su'	Beginning of models marker
40001	'nS'	
40002	1	ID
40003	66	L
...	...	...
40070	550 (as an example)	ID
40071	14	L
40072	0	DataPointA (high order word)
	120	DataPointA (low order word)
40074	16	DataPointB
40075	-3241	DataPointC
40076	2	DataPointSF
40077	-1	CtlPointSF
40078	3	CtlCount
40079	0	Pad
40080	2	CtlPointA[0]
40081	102	CtlPointB[0]
40082	2	CtlPointB[1]
40083	420	CtlPointB[1]
40084	1	CtlPointC[2]
40085	310	CtlPointC[2]
40086	0	Pad
40087	0xFFFF	End Model ID