DOF Common Types

DOF Protocol Specification

Technical Steering Committee

DOF Common Types: DOF Protocol Specification

Technical Steering Committee

Document Information

Document Tracking Number	dof-2009-1
Document Version	7.0.1, 12 January 2018
Build Version	7.0.1.8
Source Information	2018-01-12, "." at commit c4a21c8bf741a32d4fe20776861d03baab74639a on branch "master" of repository core-specifications
External Reference	DOF Common Types, OpenDOF TSC, [dof-2009-1] (7.0.1, 12 January 2018)

This document is managed by the Technical Steering Committee of the OpenDOF Project, Inc.. Contact the committee using the following information.

Contact Information

Post Mail	OpenDOF Project, Inc. Technical Steering Committee 3855 SW 153rd Drive Beaverton, OR 97003
Telephone	1-503-619-4114
Fax	1-503-644-6708
Email	tsc@member.opendof.org [mailto:tsc@member.opendof.org]

Copyright

Copyright © 2008-2018 Panasonic Corporation, Bryant Eastham

Patents

Contributors to the OpenDOF Project make a non-enforcement pledge related to any patent required to implement the protocols described in this document. For more information see https://opendof.org/pledge.

License

See https://opendof.org/license-specifications.

Table of Contents

1. About This Document 1
    1.1. Audience 1
    1.2. Authority 1
    1.3. PDU Definitions 1
        1.3.1. Context 1
        1.3.2. Generalization 2
        1.3.3. Qualifications 2
        1.3.4. Fields 4
    1.4. Specifications, Notes and Warnings 5
    1.5. Printing This Document 6
    1.6. Comparing Documents 6
        1.6.1. Text Comparison 6
        1.6.2. Graphical Comparison 6
2. Introduction 9
3. Compressed Numeric Fields 11
    3.1. Compressed Unsigned 16-Bit 11
    3.2. Compressed Unsigned 24-Bit 11
    3.3. Compressed Unsigned 32-Bit 12
4. Object Identifiers - OID 15
    4.1. Uniqueness and Comparison 17
    4.2. String Representation 18
    4.3. Object Identifier Classes 19
        4.3.1. Broadcast OID 20
        4.3.2. AS Assigned 21
        4.3.3. GUID / UUID 23
        4.3.4. ODP-TSC Assigned 24
    4.4. Object Identifier Attributes 25
        4.4.1. Session 25
5. OAP Interface Identifiers - IID 27
    5.1. IID Formatting Rules 28
    5.2. Uniqueness and Comparison 28
    5.3. Wildcard IID 28
    5.4. String Representation 29
6. Operation Identifiers 31
    6.1. Uniqueness and Comparison 32
7. DOF Types and Values 33

Specifications and PDUs

Compressed Unsigned 16-Bit `dof-2009-1-pdu-1` 11
Compressed Unsigned 16-Bit Compression is Mandatory `dof-2009-1-spec-1` 11
Compressed Unsigned 24-Bit `dof-2009-1-pdu-2` 12
Compressed Unsigned 24-Bit Compression is Mandatory `dof-2009-1-spec-2` 12
Compressed Unsigned 32-Bit `dof-2009-1-pdu-3` 13
Compressed Unsigned 32-Bit Compression is Mandatory `dof-2009-1-spec-3` 13
Object Identifier (OID) `dof-2009-1-pdu-4` 15
Object Identifier Attribute `dof-2009-1-pdu-5` 16
Attributes appear in increasing order based on Attribute Identifier. `dof-2009-1-spec-4` 17
Each Attribute Identifier appears at most once per OID. `dof-2009-1-spec-5` 17
Attribute Data may include OIDs with their own attributes `dof-2009-1-spec-6` 17
Object Identifiers are equal if a byte comparison of the PDU format (including attributes) is equal. `dof-2009-1-spec-7` 17
Object Identifiers are ordered by a byte comparison of the PDU format (including attributes), treating the bytes as unsigned numbers. `dof-2009-1-spec-8` 17
OID creation from string must allow standard string format support. `dof-2009-1-spec-10` 18
OID string format attribute ordering. `dof-2009-1-spec-12` 19
OID ASCII data escaping rules. `dof-2009-1-spec-11` 19
Registration of each OID class with the ODP-TSC must occur before its use in products. `dof-2009-1-spec-9` 20
Broadcast OID `dof-2009-1-pdu-6` 20
Multicast OID `dof-2009-1-pdu-7` 21
Multicast OIDs must include an Attribute. `dof-2009-1-spec-14` 21
AS Assigned `dof-2009-1-pdu-38` 22
GUID/UUID `dof-2009-1-pdu-14` 23
The ODP-TSC must assign the `Identifier` field before use. `dof-2009-1-spec-24` 24
ODP-TSC Assigned OID `dof-2009-1-pdu-36` 24
Registration of each OID Attribute Identifier with the ODP-TSC must occur before use in products. `dof-2009-1-spec-13` 25
Session Attribute `dof-2009-1-pdu-16` 25
Interface Identifier (IID) `dof-2009-1-pdu-21` 27
IID registries require registration with the ODP-TSC before use. `dof-2009-1-spec-17` 27
ODP-TSC must approve uses of registry 0. `dof-2009-1-spec-20` 28
Interface Identifiers except those in registry 0 and 63 require registration with the appropriate registry owner before use in product. `dof-2009-1-spec-19` 28
IIDs must use the most compact form possible. `dof-2009-1-spec-18` 28
Wildcard Interface Identifier `dof-2009-1-pdu-22` 29
IID creation from string must allow standard string format support. `dof-2009-1-spec-21` 29
Source Identifier `dof-2009-1-pdu-19` 31
Operation Identifier `dof-2009-1-pdu-20` 31
Operation Identifier Source IDs (SIDs) maximum OID BASE LENGTH is 20 bytes, the minimum is 1 byte. `dof-2009-1-spec-15` 32
Operation Identifier Source IDs (SIDs) must not have Attributes. `dof-2009-1-spec-16` 32
Each PDU must allow skipping value lists. `dof-2009-1-spec-22` 33
Type Identifier `dof-2009-1-pdu-23` 33
Type Context `dof-2009-1-pdu-39` 34
Type Reference `dof-2009-1-pdu-40` 34
Registration of Type identifiers with ODP-TSC must occur before use in products. `dof-2009-1-spec-23` 34

1. About This Document

This document contains information that is common to all DOF protocols. This is critical information for anyone who needs to work with the protocols themselves, and contains useful background information for those interested in the architecture and inner workings of any DOF implementation.

This information is highly technical. This document serves as a specification and reference against which someone could implement networked software that interoperated with DOF products without using existing libraries.

1.1. Audience

The primary audience of this document is people implementing libraries that use DOF protocols. Others that are concerned about how DOF protocols work on a network at a very low level can also benefit from this document.

Readers should be familiar with technical protocol documentation. This document is similar to an 'RFC' for DOF protocols, and familiarity with the language used in these types of documents is helpful.

This document is *not* required reading for those who need to use existing DOF libraries, although system designers may benefit from an understanding of this information. In particular, the information related to security and connections can help system designers to better understand how DOF systems work.

1.2. Authority

This document, in its English form, is the authoritative document for DOF protocols. Due to the detailed nature of protocol design, it may be difficult for a reader to implement these protocols correctly based on a translated document. When questions arise, the English version is authoritative.

All implementations that claim to be DOF compliant must satisfy the requirements set forth in this document.

This document is managed by the Technical Steering Committee of the OpenDOF Project, Inc., referred to as 'ODP-TSC'. The inside cover of this document contains contact information for the Technical Steering Committee.

The web site for the Technical Steering Committee is https://opendof.org/tsc.

1.3. PDU Definitions

This document deals with the transmission of information on a network. In order to describe this, diagrams show different fields, their network order, and positioning.

All DOF protocol documentation uses a similar diagramming style, discussed in this section. The word 'PDU', which stands for 'Protocol Data Unit', indicates these diagrams.

1.3.1. Context

Each PDU represents information sent on the network in some context. This usually means that there is information sent 'surrounding' the PDU itself. This information provides the 'context' for the PDU.

For example, the context for the DOF Protocol Stack itself is a network transport. This could be UDP/IP, TCP/IP, or any other transport. The transport will require certain headers and trailers, but they are not shown in each DOF Protocol Stack PDU. In this example the transport provide the context of the DOF Protocol Stack PDU.

In a similar way, the DOF Protocol Stack defines the context for the other DOF protocols. The Object Access Protocol, for example, can use the DOF Protocol Stack for its context. Just as the transport headers are not shown for the DOF Protocol Stack, the DOF Protocol Stack headers are not shown for the Object Access Protocol.

Understanding the context requires understanding the relative position of the different layers in the protocol stack. Each PDU defines the ordering of all of its parts, and may indicate that it is itself an instance of some other general PDU format. This means that it is always possible to start at a high-level PDU definition and understand it completely, but not always possible to identify everywhere that a particular PDU may be used (as a part of some other higher-level PDU).

Each protocol specification defines its specific contexts. The context definition includes a description of the underlying stack layer (or the containing layer), and defines the information that must be available from that layer (for reception) or passed to that layer (for transmission).

As an example, the context of the DOF Network Protocol is a transport. The transport has a requirement to provide the addresses of nodes, and so part of the context for the transport is the sender address (for reception) or the target address (for transmission). Any transport that will work with the DOF Protocol Stack must meet the context requirements.

1.3.2. Generalization

Many times a particular PDU will be a specific type of a more general PDU. In these cases, the PDU will indicate that it is an 'instance' of the more general type.

1.3.3. Qualifications

Each PDU will have qualifications placed on its use. The general qualifications include security and transport requirements. The following terms identify these qualifications. Others may be present as well.

1.3.3.1. Transport Qualifications

There are three major transport categories for DOF protocols (see the *DOF Protocol Specification*, Transport Requirements). These are None, Lossy, and Lossless. There are also three addressing types: Unicast, Multicast, and Broadcast. PDUs indicate the ability to use each of these categories as follows:

```
Session: Combinations of None, Lossy (2-node, n-node), or Lossless.
Addressing: Combinations of Unicast, Multicast, or Broadcast.
```

1.3.3.2. Security Qualifications

Each PDU categorizes security qualifications according to the major aspects of security (see the *DOF Security Specification*, Overview). These are Encryption, Data Integrity, Authentication, and Access Control. Encryption prevents viewing of the PDU, Data Integrity ensures that others do not modify the PDU, Authentication indicates knowledge of the nodes involved in the communication, and Access Control restricts those able to use the PDU.

Data Integrity is required according to the DOF Protocol Stack specification for all secure packets. Authentication is required whenever permissions are involved, or when it is important to know something

about the relationship between the sender and receiver. Encryption is required if the contents of the packet contain information that must not be visible to potential attackers.

In addition, there is the case of unsecured communication. PDUs that do not require security will list 'Unsecured' as a possible choice. Finally, certain PDUs are required to be Unsecured, and are identified as such.

Each application PDU must specify its security requirements. If Access Control is required, the PDU will indicate the permissions required for the command and for any response. However, it is typical for all of the PDUs of a given protocol to share the same requirements, and so the protocol specification may define its security qualifications in a single section. In this case, the individual PDUs should refer back to this section to avoid confusion.

It is also typical for applications to require the security of the session. A PDU indicates this as 'session security', and is its own security qualification. If the PDU allows the 'None' session type then this implies that Unsecured is allowed, as session type 'None' cannot be secure.

For PDUs that refer to session security or a common security section, one of the following formats is used:

```
Security: See the section 'Security Qualifications'.
```

or

```
Security: Session.
```

The PDU uses the following identifications to specify the security qualifications for a PDU:

```
Unsecured: (description).
Encryption: (description).
Message authentication: (description).

Permission (command): (description).
Permission (response): (description).
```

For each of these, the (description) varies. There are many different combinations of values, as described here.

Unsecured can have the values 'Required', 'Allowed', 'Not Allowed'.

Encryption can have the values 'Required', 'Allowed', 'Not Allowed'.

Message authentication can have the values 'Required', 'Allowed', 'Not Allowed'.

The specific PDU determines the permissions required (for both command and response).

There are combinations that are not valid. The following table defines all allowed combinations if a combination does not appear in the table then it is invalid.

In general, security theory links Encryption and Data Integrity. If one is 'Required' then the other must at least be 'Allowed'. Encryption and Data Integrity are the opposite of Unsecured. If either is 'Required' then unsecured must be 'Not Allowed'. Again, note that since data integrity is a common requirement for all secure packets the PDUs do not list it separately.

Unsecured / Encryption / Message Authentication	**Description**
Required / Not Allowed / Not Allowed	Indicates the PDU must be unsecured.
Allowed / Allowed / Allowed	Indicates there are no security requirements.

Unsecured / Encryption / Message Authentication	Description
Not Allowed / Allowed / Allowed	Indicates the PDU must be either encrypted or authenticated. This is the lowest level of security possible for a PDU.
Not Allowed / Required / Allowed	Indicates that encryption is required, and data integrity is optional.
Not Allowed / Allowed / Required	Indicates that encrypted is optional, but authentication is required.
Not Allowed / Required / Required	Indicates both encryption and authentication are required.

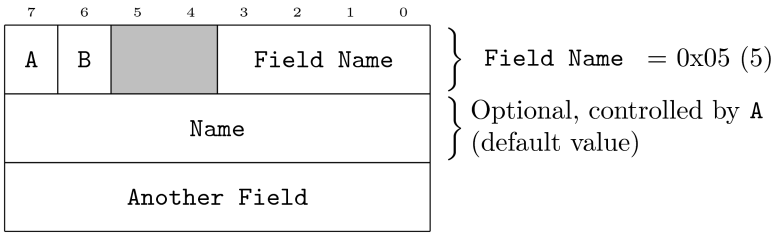
1.3.4. Fields

Each PDU shows a container, represented as a sequence of fields. The table fully describes each field.

The following is an example of a PDU description.

⇄ *Example PDU*`example-pdu-1`

PDU Context



`A`	**One bit.** Controls the presence of `Name`.
`B`	**One bit.** Field description.
`Field Name`	**Type and size description.** Field description.
`Name`	**Type and size description. Optional, controlled by** `A`. Field description.
`Another Field`	**Type and size description.** Field description.

The container represents the PDU. The diagram shows Fields first to last, left to right and top to bottom. This means that the first fields appear on the wire before the last fields. This is a general principle: things on the top and toward the left in the diagrams appear on the wire before things on the right and bottom. Note that the transport is free to use whatever ordering it must, and memory layout may different from the PDU definition. However, the transfer of a PDU from a program, over an arbitrary transport and into another program must maintain the perceived ordering between the two programs. PDU diagrams are always shown MSB first (most significant byte) and msb first (most significant bit) unless otherwise indicated.

Each field identifies its contents and provides information necessary to understand its use. In the example above, the diagram shows the first six fields of the container. Each field is named, with the exception of reserved fields which are indicated as being grayed out. Field names use `monospace text` similar to

that used in program listing. The third field shown is indicated as reserved. Reserved bits have special requirements described in the *DOF Protocol Specification*.

Several different field cases are typical, described in the following sections.

1.3.4.1. Typical Field

A typical field defines its name, its type, and its description. As shown, this field is not optional, and so it must appear in the position indicated by the diagram. The PDU uses `Field Name` throughout the related documentation to refer to the field. The type of the field is a reference to some other defined PDU. Otherwise, the PDU indicates the size of the field and any defined structure.

1.3.4.2. Optional Field

An optional field is similar to the typical case, but is indicated as optional in a note to the right. Each optional field will indicate what controls whether or not it is included as well as the default value that the field takes on when it is absent.

1.3.4.3. Fixed Values

Fields may have required values based on the specific PDU. This is common in the case of a PDU being an instance of another PDU, where the specific instance requires certain field values. In this case, it is not valid for the PDU to contain values other than that specified, if so the PDU is invalid.

The required value is shown to the right of the field as a comment.

1.3.4.4. Bit Fields

It is common for PDUs to leverage a single byte to store multiple fields (bit fields). The packing of multiple fields into a single byte is indicated as shown in the earlier example. This packing of bits may also leave space that can be indicated as reserved.

The PDU indicates reserved bits visually although the PDU does not individual name these fields.

Note that bit fields may specify default values just as other fields. However, single bit fields do not show their hexadecimal representation. In addition, bits without names that have required values may indicate their values in the place of the name.

1.4. Specifications, Notes and Warnings

This is a technical specification document. It is critical to understand how the document indicates specification items. There are also annotations for notes (implementation notes or other information that is worthy of extra notice) and warnings (pitfalls or extremely critical information). In addition, there are implementation notes, which point out optimizations that are not obvious.

Each type of callout ends either when another callout begins or when the text goes back to the normal indentation.



This is a note. The text is inset and there is a header to indicate the beginning of the text.



This is an implementation note. The text is inset and there is a header to indicate the beginning of the text.



This is a warning. The text is bold inset and there is a different header to indicate the beginning of the text.



This is an example`example-spec-1`

This is the text.

Following the summary is descriptive information about the specification item. Each specification item has a unique number assigned.

1.5. Printing This Document

The format of this document is for two-sided printing on US Letter paper.

1.6. Comparing Documents

The easiest way to compare versions of this document is to compare the corresponding source. The source format is AsciiDoctor, which is a markup-style text format. This format can be compared using any text comparison tool, including those that work with source control documents. To facilitate determining the source for each document there is information about the commit, branch, and repository along with the document version information. In addition the following guidelines are followed to make things easy to compare:

1. The input files put each sentence on a separate line, with no hard line-wrapping.

2. To the extent possible, macros and templates are used to determine formatting and structure.

3. Graphics and diagrams also use text formats.

If all you have available is a compiled document (PDF or HTML) then there are tools available to compare them. This section summarizes some of these methods with their pros and cons.

1.6.1. Text Comparison

There are several free tools that will do comparisons of the text of a PDF. One of these is the xdocdiff plugin for WinMerge. The plugin extracts the text from the PDF (including some information like page numbers), and WinMerge then compares the text.

WinMerge is available from http://winmerge.org. The xdocdiff plugin is available from http://freemind.s57.xrea.com/xdocdiffPlugin/en/index.html.

1.6.2. Graphical Comparison

In order to compare formatting, or to see the changes in the context of the document, a graphical compare is required. There are several free tools, but each has limitations.

1. diff-pdf, available from https://github.com/vslavik/diff-pdf, shows an overlay of the matching pages in different colors.

2. DiffPDF, available fromhttp://www.qtrac.eu/diffpdf.html, shows side-by-side red-lined differences of matching pages.

Both of these tools suffer from their page-by-page comparison methods. This means that when context shifts because of additions or removals that they start sensing changes at each page boundary. DiffPDF has the ability to select comparison ranges, allowing the program to synchronize at the beginning of each section, but this requires manual configuration and can be difficult for large documents.

There are also commercial tools that do an excellent job at comparing PDF documents. The first is Adobe Acrobat (http://www.adobe.com/products/acrobat.html), which in later releases has a 'Compare Document' feature. The result of the comparison is an annotated PDF, and hovering the mouse over the change will show the details.

Another commercial tool is PDF Content Comparer (http://www.inetsoftware.de/products/pdf-content-comparer) from i-net Software. This tool does side-by-side comparison, but has an intuitive auto-synchronize that keeps similar content lined up even when additions or deletions have occurred.

2. Introduction

The sections of this document describe common data types used by DOF protocols and show how to marshal those data types for use in binary network protocols. This data marshaling format is used in many DOF PDUs.

Sections specify several things for each type, including the type specification and the values that are available, the required formatting for a value of the type, and other DOF standard representations of the type and value.

This document covers both basic PDU types as well as the DOF object model data types.



The types described in this document are independent of protocol versioning. They are common among all DOF protocol versions, and are fixed. They can be added to over time, and protocols that use those new types can be negotiated if necessary according to the DOF Protocol Stack specification.

The following sections describe a type along with related PDUs and other specifications related to that type.

Throughout this document are references to 'ODP-TSC' which is the Technical Steering Committee of the OpenDOF Project, Inc.. Contact information for the ODP-TSC is at the beginning of this document (inside cover page).

3. Compressed Numeric Fields

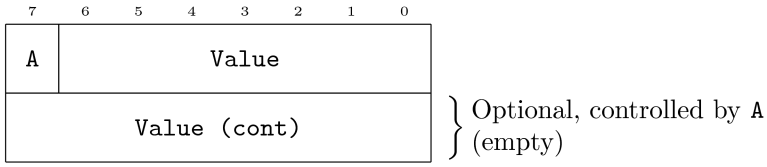
Many DOF PDUs use field compression for standard numeric types. This section presents these types.

3.1. Compressed Unsigned 16-Bit

The compressed unsigned 16-bit type can pass values between zero (0) and 2^{15}-1 (32767) inclusive.



Compressed Unsigned 16-Bit `dof-2009-1-pdu-1`



`A` **One bit.** Controls the size of `Value`.

`Value` **Either seven or fifteen bits, MSB, controlled by A.** If the `A` field is zero then the `Value` is contained in a single byte. If the `A` field is one then the `Value` is passed in fifteen bits with the least significant bits passed in a second byte.

To write a value this type, the sender compares the value with 0x80. If the value is less than 0x80, then the sender writes out a single byte containing the value. Note that a value less than 0x80 implies that `A` will be clear when written. If the value is greater than or equal to 0x80 then `A` is set and the sender writes two bytes, including `A`.

To read a value of this type, the receiver reads the first byte and `A` is checked. If `A` is zero then the value is complete. Otherwise, the next byte is read, appended, and `A` is cleared.



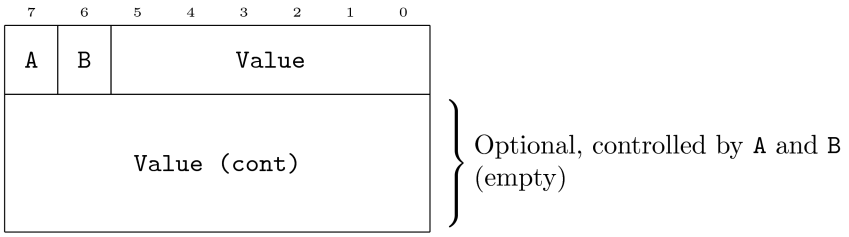
Compressed Unsigned 16-Bit Compression is Mandatory `dof-2009-1-spec-1`

The compression of the `Value` field is mandatory. Senders must pass a single byte if the value is less than 0x80.

3.2. Compressed Unsigned 24-Bit

The compressed unsigned 24-bit type can pass values between zero (0) and 2^{22}-1 (4194303) inclusive.

Compressed Unsigned 24-Bit `dof-2009-1-pdu-2`



`A` **One bit.** Controls the size of `Value` and the presence of the `B` field.

`B` **One bit.** Controls the size of `Value` or becomes part of the value.

`Value` **Either seven, fourteen, or twenty-two bits, MSB, controlled by `A` and `B`.** If `A` is zero the `Value` is seven bits and one byte is used. In this case, the `B` field is part of the `Value`. If `A` is one then the `B` field determines the size of the `Value` rather than being part of it. If `B` is zero then fourteen bits (two bytes) are used, and if `B` is one then twenty-two bits (three bytes) are used.

To write a value of this type, the sender compares the value to the following ranges and takes the specified action.

- 0x00-0x7F: Send a single byte. `A` is zero and `B` is part of the `Value`.
- 0x80-0x3FFF: Send two bytes, MSB first. `A` is one, `B` is zero, and `Value` takes the remaining space.
- 0x4000-0x3FFFFF: Send three bytes, MSB first. Both `A` and `B` are one and `Value` takes the remaining space.

To read a value of this type, the receiver does the following.

1. Read the first byte `V1`.
2. If `A` is zero, the value is `V1` and the receiver is finished.
3. Otherwise, if `B` is zero, read the next byte `V2`, compute the value as ((`V1` & 0x3F) << 8) | (`V2`), and the receiver is finished.
4. In the final case `A` and `B` are both set. Read the next two bytes (`V2`, `V3`), compute the value as ((`V1` & 0x3F) << 16) | (`V2` << 8) | `V3`, and the receiver is finished.

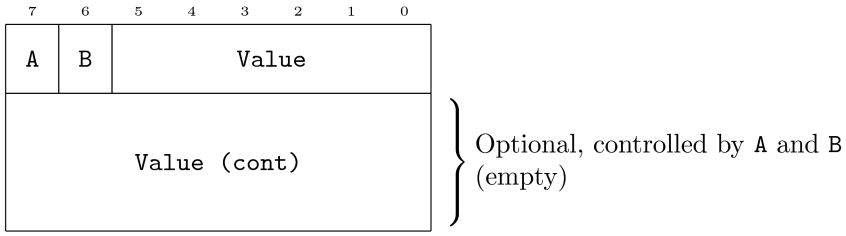
Compressed Unsigned 24-Bit Compression is Mandatory `dof-2009-1-spec-2`

The compression of the `Value` field is mandatory. There are three choices for size. Senders must use the smallest field size that can hold the value.

3.3. Compressed Unsigned 32-Bit

The compressed unsigned 32-bit type can pass values between zero (0) and 2^{30}-1 (1073741823) inclusive.

Compressed Unsigned 32-Bit `dof-2009-1-pdu-3`



`A` **One bit.** Controls the size of `Value` and the presence of the `B` field.

`B` **One bit.** Controls the size of `Value` or becomes part of the value.

`Value` **Either seven, fourteen, or thirty bits, MSB, controlled by A and B.** If `A` is zero the `Value` is seven bits and one byte is used. In this case, the `B` field is part of the `Value`. If `A` is one then the `B` field determines the size of the `Value` rather than being part of it. If `B` is zero then fourteen bits (two bytes) are used, and if `B` is one then thirty bits (four bytes) are used.

To write a value of this type, the sender compares the value to the following ranges and takes the specified action.

- 0x00-0x7F: Send a single byte. `A` is zero and `B` is part of the `Value`.
- 0x80-0x3FFF: Send two bytes, MSB first. `A` is one, `B` is zero, and `Value` takes the remaining space.
- 0x4000-0x3FFFFFFF: Send four bytes, MSB first. Both `A` and `B` are one and `Value` takes the remaining space.

To read a value of this type, the receiver does the following.

1. Read the first byte (`V1`).
2. If `A` is zero, the value is `V1` and the receiver is finished.
3. Otherwise, if `B` is zero, read the next byte (`V2`), computer the value as ((`V1` & 0x3F) << 8) | (`V2`), and the receiver is finished.
4. In the final case `A` and `B` are both set. Read the next three bytes (`V2`, `V3`, and `V4`), computer the value as ((`V1` & 0x3F) << 24) | (`V2` << 16) | (`V3` << 8) | `V4`, and the receiver is finished.

Compressed Unsigned 32-Bit Compression is Mandatory `dof-2009-1-spec-3`

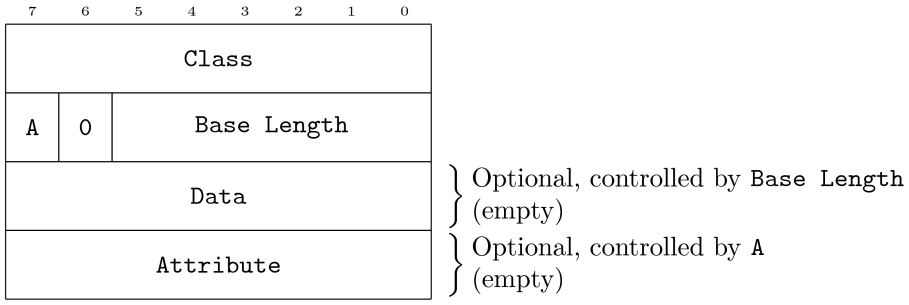
The compression of the `Value` field is mandatory. There are three choices for size. Senders must use the smallest field size that can hold the value.

4. Object Identifiers - OID

Object identifiers are globally unique identifiers that DOF protocols and implementations use. This can be confusing due to the multiple uses of 'object' in DOF libraries. In the case of object identifiers, the use of 'object' is in the broadest sense. Examples are authentication identities, security identifiers, and object models. A principle of an object identifier is that it is globally unique without requiring centralized registration of each identifier. This means that the uniqueness comes from the data itself, guaranteed because the format provides that different types of data do not 'alias' each other. In reality, this usually means that the registration requirements apply to the OID data. For example, an OID that uses a MAC address relies on the registration of the OID as well as the manufacturer keeping track of the individual MAC address assignments.

The following general PDU applies to all object identifiers.

Object Identifier (OID) `dof-2009-1-pdu-4`



`Class` **Instance of** Compressed Unsigned 32-Bit`dof-2009-1-pdu-3`.The OID class, registered with ODP-TSC.

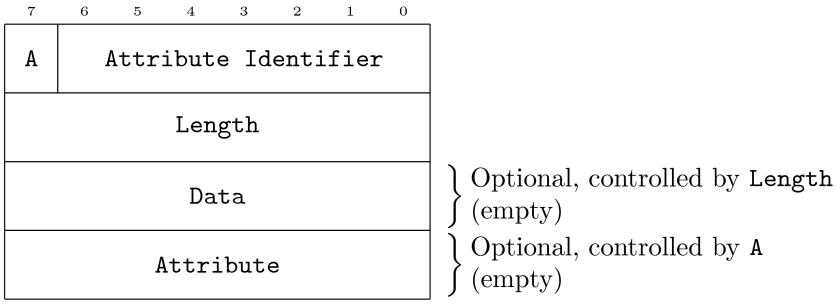
`A` **One bit.** Controls the presence of `Attribute`.

`Base Length` **Six bits.** This contains the length of the `Data`.

`Data` **Variable bytes.** The data associated with the OID. `Base Length` bytes.

`Attribute` **Instance of** Object Identifier Attribute`dof-2009-1-pdu-5`.**Optional, controlled by** `A`. If `A` is one then this field is present. Attributes form an ordered list, with each attribute containing a flag controlling the presence of the next attribute.

Object Identifier Attribute `dof-2009-1-pdu-5`



`A`	**One bit.** Controls the presence of `Attribute`.
`Attribute Identifier`	**Seven bits.** The attribute type, registered with ODP-TSC.
`Length`	**One byte.** This field contains the length of the `Data`.
`Data`	**Variable bytes.** The data associated with the attribute. `Length` bytes.
`Attribute`	**Instance of** Object Identifier Attribute`dof-2009-1-pdu-5`.**Optional, controlled by** `A`. If `A` is one then this field is present. Attributes form an ordered list, with each attribute containing a flag controlling the presence of the next attribute.

Note that in some cases the attribute `Data` may be another OID. In this case the `Length` represents the entire encapsulated OID (which itself also includes a length and possibly other nested attributes). It is possible to create OIDs that are too long to use as `Data`.

Attributes chain one after the other until indicated by an attribute with `A` equal to zero.



`A` indicates the presence of another attribute following the one in which the bit appears.

The OID and attribute formats are independent of the data formatting that they contain. This requires additional length information, but means that implementations do not need to understand every OID class and attribute identifier.

The use of the different bits in the header combined with the definition of the length byte means that to read an object identifier requires several decisions:

1. Determine the OID `Class`, which is compressed.
2. Determine the OID `Base Length` and whether there is an `Attribute`.
3. Determine the OID `Data`.
4. If there is an `Attribute`, determining it along with any further attributes.

Any program that uses object identifiers should assume that they are binary and opaque unless they have some knowledge of the particular OID `Class`.

The OID `Class` defines the format of the OID `Data`. The `Class` may or may not require a particular length, and object identifiers (even with the same class) may have different lengths.

Attributes appear in increasing order based on `Attribute Identifier`. **`dof-2009-1-spec-4`**

This guarantees byte-comparisons of OIDs.

Each `Attribute Identifier` appears at most once per OID. **`dof-2009-1-spec-5`**

This ensures that there is no ambiguity in the comparison of OIDs. This also means that there are a maximum of 128 different attributes attached to single OID.

This limits the absolute worst-case maximum length of an OID to 5 + 63 + 128 * (257) or 32964 bytes. However, this limit is unreachable because most attributes define their own maximum length to be much less than 255.

Attribute `Data` may include OIDs with their own attributes **`dof-2009-1-spec-6`**

These nested OIDs may include an `Attribute Identifier` present in the 'parent' OID, but the attribute lists are logically distinct and so this does not violate either dof-2009-1-spec-4 or dof-2009-1-spec-5.

4.1. Uniqueness and Comparison

All object identifiers uniquely represent their objects. There are several ways to guarantee uniqueness in distributed systems. The best way for OIDs is to base the object identifier on a physical property of the object it represents (like a product serial number). A second choice is to base the object identifier on a networking property associated with the object (like a MAC address or an email address), but in this case it is critical in most cases that the property is static, so that the association of object identifier to object does not change over time.

DOF implementations define both equality and ordering for OIDs.

Object Identifiers are equal if a byte comparison of the PDU format (including attributes) is equal. **`dof-2009-1-spec-7`**

Implementations may choose to store OIDs in other formats.

Object Identifiers are ordered by a byte comparison of the PDU format (including attributes), treating the bytes as unsigned numbers. **`dof-2009-1-spec-8`**

Implementations may choose to store OIDs in other formats, but must enforce this sort order. The formatting of OIDs means that longer OIDs will differ from shorter OIDs in a byte position before the last byte of the shorter OID.



OID equality and comparison have the same definition as memcmp() if their data is treated as a byte array. Note that the guarantee of difference before the last byte of the shorter OID ensures that in comparison the length of either OID may be used. This means that additional length comparisons (usually required for memcmp()) are not necessary for OIDs.

4.2. String Representation

There are obviously many different possible ways of representing an OID as a string. It is useful to provide a core definition that is universally supported. This definition is not meant to replace other possible representations, some of which may be much more appropriate in some cases.

There are two cases supported: all hex and formatted. The all hex format is an exact representation of the protocol-defined OID, with the hex string defining the OID. This means that the value must include all special bits, lengths, etc. The formatted format allows for fields (like lengths) and continuation bits to be determined automatically.

Implementations that allow textual OID specification must support the standard format. They may support additional formats as well. The standard format may be used for human-readable output, although this is not its primary purpose.



OID creation from string must allow standard string format support. **`dof-2009-1-spec-10`**

Implementations that allow for OIDs to be created from strings must support the standard format. Other formats may be allowed, but they must not begin with `[` or `{` so as not to conflict with the standard format.

In the following definitions, fields that are **`bold italic`** are literals. The 'standard format' is the OID field (defined below). Note that the OID must begin with either a `[` or a `{`. All non-standard formats are prohibited from starting with these two characters.

```
Nibble: [ 0-9, a-f, A-F ] Byte: Nibble Nibble Digit: [ 0-9 ] Number: Digit
+ Sep: space | colon | dash | empty Block: Byte { Sep Byte }*

Hex-field: { Block } String-field: [ US-ASCII string (defined below) ]
Field: Hex-field | String-field

Hex-OID: Hex-field

AVP: ( Number | Hex-field ) : Field | OID AVP-Cont: | AVP-List | empty
AVP-List: AVP AVP-Cont AVP-Block: ( AVP-List ) | empty

Format-OID: [ ( Number | Hex-field ) : Field AVP-Block ]

OID: Hex-OID | Format-OID
```

Regarding `String-field`, each ASCII character is significant (spaces must be treated as part of the data). Only characters between 32 (inclusive) and 126 (inclusive) are allowed. The following special sequences are defined. These are allowed anywhere in `String-field`, and depending on position may be required in order to prevent terminating the `String-field` or interpreting a `String-field` as a `Hex-field`. In each case the sequence represents the second character in the escape sequence. Other characters are not allowed in an escape sequence.

- \\ is used to include a single backslash character.

- \{ is used for {
- \} is used for }
- \[is used for [
- \] is used for]
- \(is used for (
- \) is used for)
- \| is used for |

OID string format attribute ordering. `dof-2009-1-spec-12`

The ordering of attributes in the string must match that defined by the OID specification; no reordering of attributes is allowed. Incorrect ordering is not allowed and must result in a failure to understand the string format.

Note that it is possible to use this format to encode all valid OIDs. However, it is also possible to define illegal or malformed OIDs as well. In particular:

- The ranges of numeric values can exceed those allowed by the specification for a particular class. The library will check overall numeric ranges that apply to the type itself.
- Specifications of individual OID types regarding lengths and data formatting are not defined.

This means that once the fields defined are understood they must be further validated by the application in order to be fully correct.

The `AVP-Block` must define attributes in the order defined by the OID specification (increasing and without duplicates).

OID ASCII data escaping rules. `dof-2009-1-spec-11`

The standard string format defines character escapes for ASCII data. Only the defined escape characters are allowed.

4.3. Object Identifier Classes

OID classes uniquely identify the data used in the OID and ensure uniqueness when the OID data itself may not be entirely unique (it other words, it may alias to some other OID data).

The class uniquely identifies the data associated with the object identifier. Classes do not represent the type of the OID (implying something about what the OID represents). For example, it is acceptable for a class based on a network parameter (like a MAC address) to be used as a security identity. There are cases, however, when implementations may assume things about the use of the OID based on its class (see, for example, the Broadcast class).

This document does not contain a comprehensive list of all OID classes; although this section contains classes referenced in other DOF protocols. The ODP-TSC manages the assignment of OID classes.

Applications that interact with users directly may choose to translate OID classes into human-readable text. While this is possible, this is not the primary use of an OID. Some OID classes may be appropriate in this context, but others are not.

For a current list of registered classes, please contact the ODP-TSC or go to https://opendof.org/registry-oid-class.

Registration of each OID class with the ODP-TSC must occur before its use in products. `dof-2009-1-spec-9`

This ensures that each OID will be unique.

4.3.1. Broadcast OID

One important use of the OID is in the DOF object model, where it represents an 'object' as a provider of functionality. Since these 'objects' can be the targets of operations in DOF systems, it is useful to have an OID that represents 'all' and 'some' different OIDs.

The Broadcast OID is used for the 'all' case, and in the DOF object model represents all possible targets. When augmented with the group attribute (described later) it takes on a multicast meaning, where it represents all possible targets within a group.

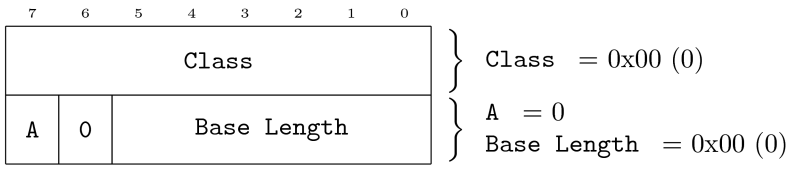
In situations that do not imply a 'target' this OID has no special meaning.

4.3.1.1. Data Format - Broadcast

This OID has no data. In order to be 'globally unique' it must take on the meaning of broadcast, since it has no data with which to differentiate objects.

Broadcast OID `dof-2009-1-pdu-6`

Instance of Object Identifier (OID)`dof-2009-1-pdu-4`



`Class` **Instance of** Compressed Unsigned 32-Bit`dof-2009-1-pdu-3`.The OID class, registered with ODP-TSC.

`A` **One bit.**

`Base Length` **Six bits.**

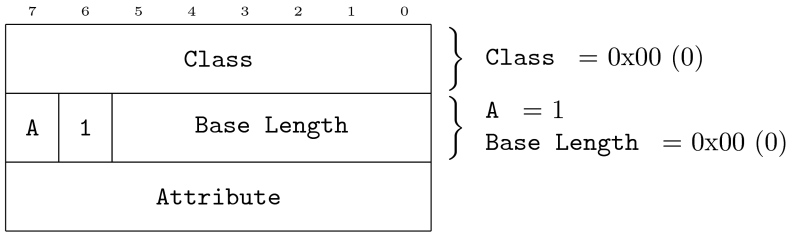
4.3.1.2. Data Format - Multicast

The multicast data format is technically the same as the broadcast format, with the exception being that an attribute (defined below) is required. When used to represent a target in the DOF object model the use

of group attribute represents all targets that share the same group attribute value. This logic uses an 'at least' comparison, meaning that the attributes present are required to match, but the target may contain additional attributes.

Multicast OID `dof-2009-1-pdu-7`

Instance of Object Identifier (OID)`dof-2009-1-pdu-4`



`Class` **Instance of** Compressed Unsigned 32-Bit`dof-2009-1-pdu-3`.The OID class, registered with ODP-TSC.

`A` **One bit.**

`Base Length` **Six bits.**

`Attribute` **Array of** Object Identifier Attribute`dof-2009-1-pdu-5`.There is always at least one Attribute present. Attributes form an ordered list, with each attribute containing a flag controlling the presence of the next attribute.

Multicast OIDs must include an Attribute. `dof-2009-1-spec-14`

The combination of a zero CLASS, zero BASE LENGTH, and an attribute defines multicast in the DOF object model.

4.3.2. AS Assigned

The Authentication Server can generate object identifiers as discussed in the DOF Security Specification. These object identifiers are somewhat different in that they are not universally unique – they are only unique within the security domain that assigns them.

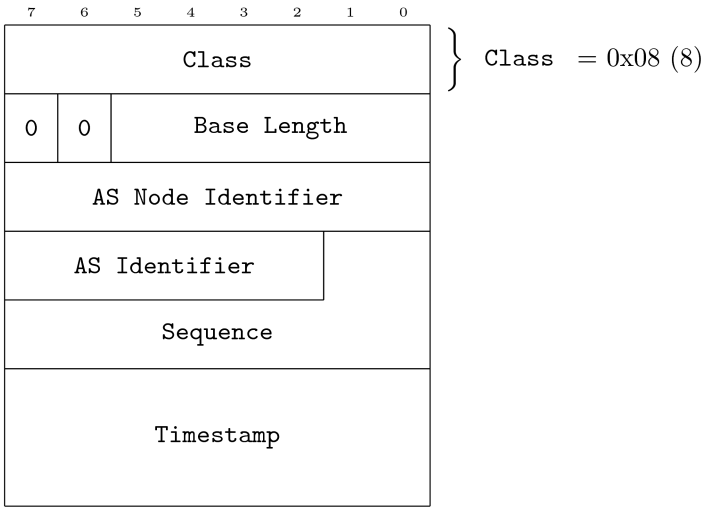
The lack of global uniqueness means that this class must be used only as a source identifier as discussed in the DOF Protocol Specification.

4.3.2.1. Data Format

The data format is unique within a security domain through a combination of information known to the AS that is associated with a credential. This credential-based information, combined with information based on time and sequence of creation, provide uniqueness within the scope of the security domain.

AS Assigned `dof-2009-1-pdu-38`

Instance of Object Identifier (OID)`dof-2009-1-pdu-4`



`Class` **Instance of** Compressed Unsigned 32-Bit`dof-2009-1-pdu-3`.The OID class, registered with ODP-TSC.

`Base Length` **Six bits.** This contains the sum of the `AS Node Identifier`, `AS Identifier`, `Sequence`, and `Timestamp` fields and is at least 0x07 (7) with a maximum length of 0x0A (10).

`AS Node Identifier` **One to four bytes.** Contains the node identifier assigned by the AS to the credentials used to generate the OID. The format must not have leading zeroes, although the minimum size allows a single zero byte. The number of bytes in this field will be equal to `Base Length` minus six.

`AS Identifier` **Six bits.** The identifier of the AS front end that is generating the OID. Each AS front end for a given security domain must use a different AS identifier.

`Sequence` **Ten bits.** This field is split between the two bytes, passed with the most significant bits first. It represents the sequence of the OID assignment by the AS in the specified time unit indicated by

`Timestamp` **Four bytes.** The timestamp of the generation of the OID in seconds, using the standard UNIX time (based January 1, 1970). The timestamp will wrap, but only after many years. The wrapping does not cause a problem. After 68 years, a duplicate assignment is possible.

4.3.3. GUID / UUID

A UUID is a standardized unique data value. The tradeoff with a UUID is the size that it requires. However, since it is globally unique systems can use it to create an OID. Designs should avoid the use of this class for two reasons:

1. The size is very large (16 bytes of data) as compared to other classes.

2. The format is random, making it more difficult to manage as compared to classes based on a temporally fixed physical or logical property.

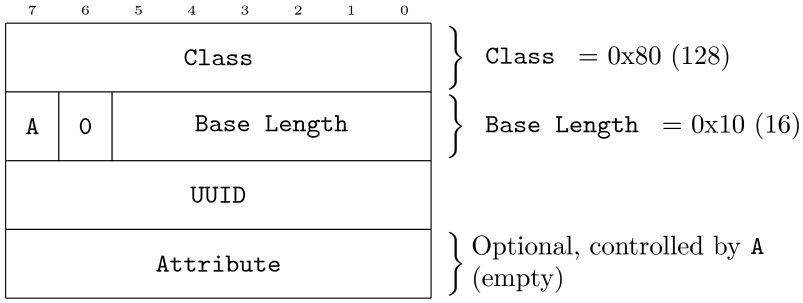
However, in cases where a random OID is acceptable, this format is appropriate.

4.3.3.1. Data Format

The following format defines the UUID class. The definition of UUID ensures global uniqueness of this OID class.

GUID/UUID `dof-2009-1-pdu-14`

Instance of Object Identifier (OID)`dof-2009-1-pdu-4`



`Class` **Instance of** Compressed Unsigned 32-Bit`dof-2009-1-pdu-3`.The OID class, registered with ODP-TSC.

`A` **One bit.** Controls the presence of `Attribute`.

`Base Length` **Six bits.** This contains the length of the `UUID`.

`UUID` **Sixteen bytes.** The UUID address, MSB first. See RFC 4122 for more information. Note that the rule for MSB first includes all fields. This is not the same as the typical UUID storage format (which is dependent on machine endianness). `Base Length` bytes.

`Attribute` **Instance of** Object Identifier Attribute`dof-2009-1-pdu-5`.**Optional, controlled by** `A`. If `A` is one then this field is present. Attributes form an ordered list, with each attribute containing a flag controlling the presence of the next attribute.

4.3.4. ODP-TSC Assigned

There are situations where no good unique identifier exists, and implementers do not desire the UUID format because of its large size. In this case, the ODP-TSC can assign a unique identifier used with this class. The recipient then can use all OIDs that use that identifier as its base – giving relatively small size OIDs based on the assigned identifier and additional unique data.

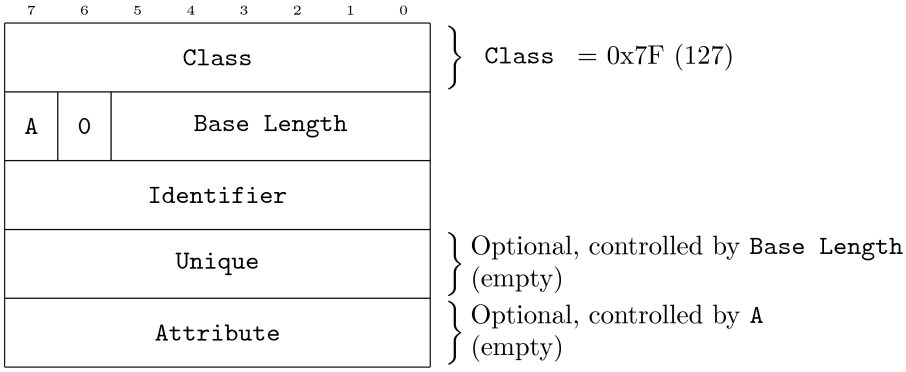
For a current list of registered `Identifier` assignments, please contact the ODP-TSC or go to https://opendof.org/registry-oid-class-odp-tsc.

The ODP-TSC must assign the `Identifier` field before use. `dof-2009-1-spec-24`

The ODP-TSC must assign each ODP-TSC Assigned `Identifier` before use in products.

ODP-TSC Assigned OID `dof-2009-1-pdu-36`

Instance of Object Identifier (OID)`dof-2009-1-pdu-4`.



`Class` **Instance of** Compressed Unsigned 32-Bit`dof-2009-1-pdu-3`.The OID class, registered with ODP-TSC.

`A` **One bit.** Controls the presence of `Attribute`.

`Base Length` **Six bits.** This contains the length of the `Identifier` and `Unique` fields. It must be at least 1.

`Identifier` **Instance of** Compressed Unsigned 32-Bit`dof-2009-1-pdu-3`.The ODP-TSC-assigned identifier, passed as a compressed number.

`Unique` **Variable length.** Optional data associated with the OID that uniquely identifies objects that share the same `Identifier`. This is an object index, MSB first. The field must not use leading zero bytes.

`Attribute` **Instance of** Object Identifier Attribute`dof-2009-1-pdu-5`.**Optional, controlled by** A. If A is one then this field is present. Attributes form an ordered

list, with each attribute containing a flag controlling the presence of the next attribute.

4.4. Object Identifier Attributes

Attribute identifiers uniquely identify the data used in the Attribute of an OID.

This document does not contain a comprehensive list of all Attribute Identifiers. The ODP-TSC manages the assignment of Attribute Identifiers.

For a current list of registered attribute identifiers, please contact the ODP-TSC or go to https://opendof.org/registry-oid-attribute.

Registration of each OID Attribute Identifier with the ODP-TSC must occur before use in products. `dof-2009-1-spec-13`

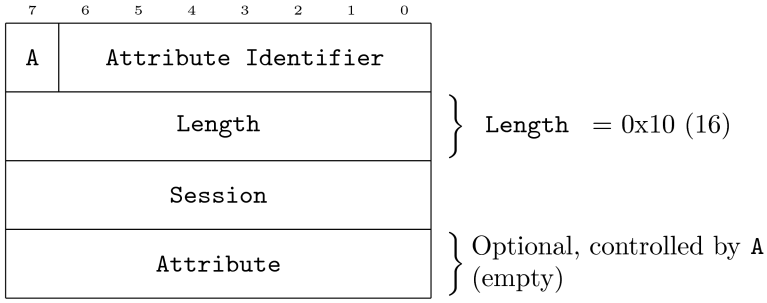
This ensures unique assignment and compatibility.

4.4.1. Session

This attribute identifies a session. It augments the OID of the bindings provided in the session. The value is a UUID/GUID in order to ensure uniqueness.

Session Attribute `dof-2009-1-pdu-16`

Instance of Object Identifier Attribute`dof-2009-1-pdu-5`



`A`	**One bit.** Controls the presence of `Attribute`.
`Attribute Identifier`	**Seven bits.** The attribute type, registered with ODP-TSC.
`Length`	**One byte.** This field contains the length of the `Session`.
`Session`	**Sixteen bytes.** The UUID address, MSB first. See RFC 4122 for more information. Note that the rule for MSB first includes all fields. This is not the same as the typical UUID storage format (which is dependent on machine endianness).

`Attribute`	**Instance of** Object Identifier Attribute$_{\texttt{dof-2009-1-pdu-5}}$.**Optional, controlled by** `A`. If `A` is one then this field is present. Attributes form an ordered list, with each attribute containing a flag controlling the presence of the next attribute.

5. OAP Interface Identifiers - IID

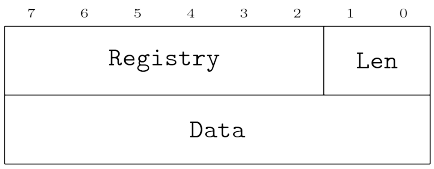
Each OAP interface has a globally unique identifier (Interface Identifier, or IID) associated with it. These identifiers are a critical aspect of the DOF object model used in OAP. Each IID represents a collection of properties, methods, exceptions, and events.

There can be no conflicts between registered interfaces, and so each registered interface identifier is unique among all systems. DOF systems interoperate, and so designs must assume that any two networks may at some point connect. Any conflict in the interface identifiers used in those systems would cause many problems in the applications that use them.

To solve the problem of uniqueness, the format of the interface identifier is very flexible. For testing and dynamic generation there are special registries. All final products must use registered interface identifiers because they are much smaller (down to two bytes).

The following figure illustrates the format of an interface identifier on the wire.

Interface Identifier (IID) `dof-2009-1-pdu-21`



`Registry` **Six bits.** The registry of the IID, registered with the ODP-TSC.

`Len` **Two bits.** Defines the length of the `Data`, which is $2^{(LEN-1)}$. `Len` equal to zero (0) is invalid.

`Data` **Variable length, controlled by `Len`.** The identifier for the IID within the `Registry`.

Registry 63 (sixty-three) is uncontrolled, and can be used during development and testing. There are no guarantees of uniqueness in registry 63. Interfaces in this registry are not registered.

The registry zero (0) is reserved for situations where a dynamic interface is required. There is no registration for these dynamic interfaces, and so conflicts may exist. The potential for confusion is resolved by requiring that only the provider of the interface in registry zero is allowed to provide a definition, as discussed in the Object Access Protocol (V1) specification. Uses of this registry require approval by the ODP-TSC.

For a current list of assigned registries, please contact the ODP-TSC or go to https://opendof.org/registry-iid-registry.

IID registries require registration with the ODP-TSC before use. `dof-2009-1-spec-17`

All IID Registries require registration with the ODP-TSC before use in products.

ODP-TSC must approve uses of registry 0. `dof-2009-1-spec-20`

Registry 0 is not a convenient way to bypass other registration requirements. The ODP-TSC must review and approve any design using registry 0 before use in a product.

Interface Identifiers except those in registry 0 and 63 require registration with the appropriate registry owner before use in product. `dof-2009-1-spec-19`

This enforces the registration process, review, and documentation. It also prevents conflicting assignments.

5.1. IID Formatting Rules

The formats described above allow for multiple representations of the same interface identifier with the addition of zero padding. In order to avoid this, and to provide that binary comparisons of interface identifiers yield correct results, interface identifiers use the most compact representation possible.

IIDs must use the most compact form possible. `dof-2009-1-spec-18`

An interface identifier must always be passed in the most compact form (meaning the lowest value of LEN) possible.

Although not mandated, the ODP-TSC recommends that registries preserve the smaller `Len` values for interfaces that are frequently used. As there are only 256 IIDs in each registry with a `Len` value of 1, they should be granted carefully.

5.2. Uniqueness and Comparison

As mentioned above, all registered interface identifiers are globally unique. The use of registries allows small (down to 1 byte) data values to be unique (not to collide to any other). It is a critical error to have a collision where two different registered interfaces use the same IID, or where the definitions of the same registered IID are different. Extreme care is required to avoid these problems, which are common during development.

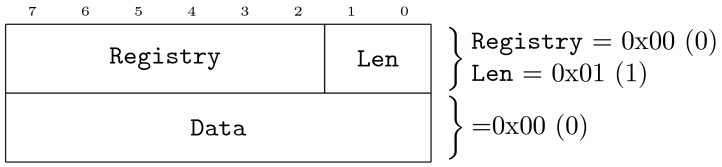
Comparison of IIDs is byte-by-byte, up to the length of the shorter interface. DOF implementations define both equivalence (identical length, identical bytes) and ordering. The definition of ordering is by comparison of bytes. Larger byte values sort after smaller values. If two identifiers have equal bytes up to the end of one of them, then the longer IID sorts after the shorter.

5.3. Wildcard IID

There is a single IID defined as a wildcard. Its IID and definition are global, even though it uses registry zero (0).

Wildcard Interface Identifier `dof-2009-1-pdu-22`

Instance of Interface Identifier (IID)`dof-2009-1-pdu-21`



`Registry` **Six bits.** The registry of the IID, registered with the ODP-TSC.

`Len` **Two bits.** Defines the length of the `Data`, which is $2^{(LEN-1)}$. `Len` equal to zero (0) is invalid.

`Data` **Variable length, controlled by `Len`.** The identifier for the IID within the `Registry`.

5.4. String Representation

There are many different possible ways of representing an IID as a string. It is useful to provide a core definition that is universally supported. This definition is not meant to replace other possible representations, some of which may be much more appropriate in some cases.

There are two cases supported: all hex and formatted. The all hex format is an exact representation of the protocol-defined IID, with the hex string defining the IID. This means that the value must include all special bits, lengths, etc. The formatted format allows for fields (like lengths) to be determined automatically.

Implementations that allow textual IID specification must support the standard format. They may support additional formats as well. The standard format may be used for human-readable output, although this is not its primary purpose.

IID creation from string must allow standard string format support. `dof-2009-1-spec-21`

Implementations that allow for IIDs to be created from strings must support the standard format. Other formats may be allowed, but they must not begin with `[` or `{` so as not to conflict with the standard format.

In the following definitions, fields that are **`bold italic`** are literals. The “standard format” is the IID field (defined below). Note that the IID must begin with either a `[` or a `{`. All non-standard formats are prohibited from starting with these two characters (this will be a spec item).

```
Nibble: [ 0-9, a-f, A-F ] Byte: Nibble Nibble Digit: [ 0-9 ] Number: Digit
+ Sep: space | colon | dash | empty Block: Byte { Sep Byte }*

Hex-field: { Block }

Hex-IID: Hex-field

Format-IID: [ ( Number | Hex-field ) : ( Number | Hex-field ) ]
```

```
IID: Hex-IID | Format-IID
```

The `Hex-IID` format is taken as a literal sequence of bytes that make up the IID. In the `Format-IID` case the `Hex-field` values are treated as numbers (represented in hex). This means that the formatting requirements of the IID (for example, no leading zero bytes) are not enforced in the string format. Conversion to the actual bytes from the `Format-IID` format would then take care of ensuring the correct byte formatting.

Note that it is possible to use this format to encode all valid IIDs. However, it is also possible to define illegal or malformed IIDs as well. In particular:

- The ranges of numeric values can exceed those allowed by the specification.

This means that once the fields defined are understood they must be further validated by the application in order to be fully correct.

6. Operation Identifiers

Operation identifiers are globally unique identifiers that correspond to commands and responses between nodes. The section on the DOF Protocol Stack discusses them in more detail.

Operation identifiers, or OPIDs, are global, which means that all protocols and all protocol versions transmit and compare them identically.

In order to accomplish this requirement, this section defines the general structure of the OPID. All protocols work with this format, although they may transmit fields using other formats.

Each OPID contains two pieces of information, the source identifier, or SID, and the operation count, or OPCNT. The usage of these terms is universal in DOF implementations, similar to OIDs and IIDs.

This universal nature of operations means two things:

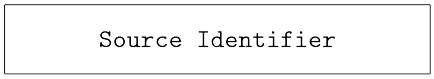
- Operation identifiers (OPIDs) are globally unique, which allows for detection of loops.
- Operation identifiers (OPIDs) are transport independent. To the extent possible, all DOF protocol stack behavior is consistently defined, and is independent of the transport that the PDU arrives on.

The following formats define operation identifiers.



Source Identifier `dof-2009-1-pdu-19`

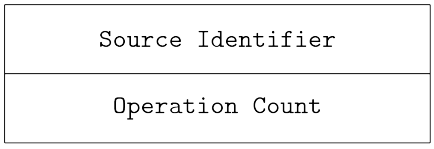
Instance of Object Identifier (OID)`dof-2009-1-pdu-4`



`Source Identifier` **Instance of** Object Identifier (OID)`dof-2009-1-pdu-4`.The maximum `Base Length` is 0x14 (20) bytes, with a minimum of one byte. The `A` field is zero (0).



Operation Identifier `dof-2009-1-pdu-20`



`Source Identifier` **Instance of** Source Identifier`dof-2009-1-pdu-19`.

`Operation Count` **Instance of** Compressed Unsigned 32-Bit`dof-2009-1-pdu-3`.The unique identifier of the operation in the context of the source.

This definition is not a wire format. The specific methods used to pass OPIDs on the wire are specific to the protocols used.

Operation Identifier Source IDs (SIDs) maximum OID BASE LENGTH is 20 bytes, the minimum is 1 byte. `dof-2009-1-spec-15`

This requirement allows small MTU transports to work with OPIDs from other, larger MTU transports. This length does not include the class, or length. Since the SID must ensure uniqueness, it must not use the Broadcast OID.

Operation Identifier Source IDs (SIDs) must not have Attributes. `dof-2009-1-spec-16`

This requirement allows small MTU transports to work with OPIDs from other, larger MTU transports.

6.1. Uniqueness and Comparison

All operation identifiers uniquely represent their operations. Comparison of OPIDs requires three steps:

1. Convert the OPIDs into the standard format. This means to expand the SID if implied, and convert the OPCNT to a full 30 bits.
2. Compare the SIDs according the rules for comparison of OIDs.
3. Compare the OPCNTs as unsigned integers.

7. DOF Types and Values

DOF implementations define a common method for defining and exchanging type and value information. Any DOF application protocol may use these definitions. It is important to note that the DOF type system is extendable, and so new basic types may be added in the future. This means that an existing application protocol may receive PDUs that refer to these new basic types, and the application defines how it will handle this information (which it may not understand). In order to make it possible to skip unrecognized type definitions, each type definition follows a fixed format.

There is not a common format for values as there is for types. This means that if a reader does not understand a value in a list, then it cannot read further values.

Each PDU must allow skipping value lists. `dof-2009-1-spec-22`

There are two ways to meet this requirement. The easiest is to ensure that values are always at the end of the PDU, which makes it possible to skip by definition (go to the end of the PDU). The second is to include a length in the PDU. This would only be required if additional fields were required after the values.

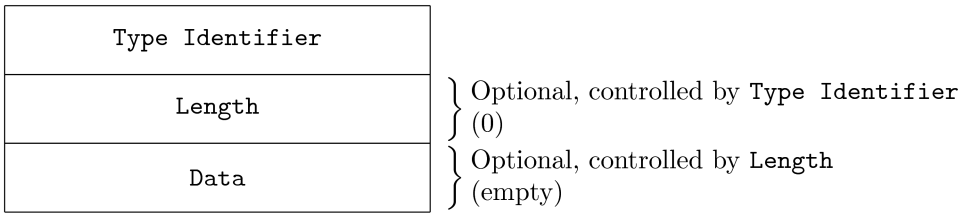
Types may reference other types (for example, a structure has types for its fields). These references may form complex relationships between different types, including back references (forming loops in the types).

In order to provide a common framework for allowing these type references, a common format called a type context encapsulates all type definitions. The context assigns an order to the type definitions, and allows the use of an index by other types to form references. The type context is self-contained, meaning that it contains all types directly. The indexes may be either forward references or backward references. However, all loops in type definitions must be breakable (otherwise, it would be impossible to create a value that represented the type).

In order to create a self-referencing type it must be possible to declare a type without providing its complete content. Types that the context declares and then later defines are stored as a reference in the context. This is the way that loops can be stored and read back in a way that preserves the structure of the type declaration.

Each type definition follows the same basic format:

Type Identifier `dof-2009-1-pdu-23`

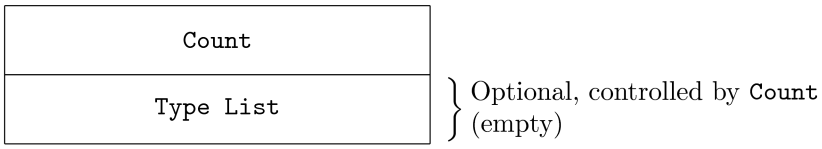


`Type Identifier` **Instance of** Compressed Unsigned 16-Bit`dof-2009-1-pdu-1`.Controls the presence of `Length`. If this value is odd then the `Length` field is present. This identifies each basic type, registered with ODP-TSC.

`Length`	**Instance of** Compressed Unsigned 16-Bit`dof-2009-1-pdu-1`.Controls the presence of `Data`.
`Data`	**Variable length. Optional, controlled by** `Length`. The type data depends on the `'name': 'Type Identifier'}`. This data can provide additional information related to the type.

The type context has the following structure:

Type Context `dof-2009-1-pdu-39`

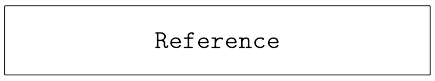


`Count`	**Instance of** Compressed Unsigned 16-Bit`dof-2009-1-pdu-1`.Controls the length of `Type List`.
`Type List`	**Array of** Type Identifier`dof-2009-1-pdu-23`.**Optional, controlled by** `Count`. A list of type definitions. The list starts with index zero, and each type in the list has an index one greater than the preceding type.

The type context includes the definitions of all types. Each index in the type context defines a unique type.

Type Reference `dof-2009-1-pdu-40`

Instance of Compressed Unsigned 16-Bit`dof-2009-1-pdu-1`.



`Reference`	**Instance of** Compressed Unsigned 16-Bit`dof-2009-1-pdu-1`.This identifies the referenced type by index in the associated type context.

External references to types require a type reference and a type context in order to be resolved.

For a current list of registered type identifiers, please contact the ODP-TSC or go to https://opendof.org/registry-types.

Registration of Type identifiers with ODP-TSC must occur before use in products. `dof-2009-1-spec-23`

This includes the definitions of the types and the wire format and meaning for each type. After registration, the formatting and meaning of a type must not change.

DOF value marshaling controls how data values of different types are put on the wire. Marshaled data can take two forms: single values (including single compound types like structures and arrays) or lists of values. In the same way, type specifications can take the same two forms.

In general, PDUs will include either type specifications or values, but not both. This means that the receiver has to know the type specifications in order to retrieve values from the wire.

Note that the type specifications and values passed on the wire represent only the information necessary to get the data from one node to another, and do not include higher-level semantic information (like minimum/maximum values or enumeration information). DOF implementations assume that the provider and requestor have already agreed on the semantic meaning of the values through other means.

For each type available there are two formats specified. The first is the type specification format; the second is the data format.