USDC File Format Specification

- Introduction
  - Versions
    - Version History
  - Order of Reads
  - Integers
  - Index
- Compression
  - LZ4 Compression
  - Integer Compression
- Preamble
  - Bootstrap
  - Table of Contents
  - Sections
    - Tokens
    - Strings
    - Fields
    - Field Sets
    - Paths
    - Specs
- Constructing the Layer
  - Layer Metadata
  - Prims
  - Variant Sets
  - Variants
  - Properties (Attribute)
- Value Representations

    - Inlined Types
    - Singular Offset Types
    - Array Offset Types
        - Compressed Integer Arrays
        - Compressed Floating Point Arrays
- Value Types
    - Type Table
    - Base Types
        - Bool
        - UChar
        - Int
        - Uint
        - Int64
        - UInt64
        - Half
        - Float
        - Double
        - String
        - Token
        - AssetPath
        - Payload
        - ValueBlock
        - Value
        - UnregisteredValue
        - TimeCode
- Other Types
    - References
    - Layer Offset
    - Quaternions
        - QuatD
        - QuatF
        - QuatH
    - Vectors (Mathematical)
        - Vec2d
        - Vec2f

        - Vec2h
        - Vec2i
        - Vec3d
        - Vec3f
        - Vec3h
        - Vec3i
        - Vec4d
        - Vec4f
        - Vec4h
        - Vec4i
    - Matrix
        - Matrix2d
        - Matrix3d
        - Matrix4d
    - Dictionary
    - List Operations
        - Header
        - Contents
        - TokenListOp
        - StringListOp
        - ReferenceListOp
        - IntListOp
        - Int64ListOp
        - UIntListOp
        - UInt64ListOp
        - PayloadListOp
        - UnregisteredValueListOp
    - Vectors (Arrays)
        - PathVector
        - TokenVector
        - DoubleVector
        - LayerOffsetVector
        - StringVector
    - Specifier
    - Permission

- Variability
- Variant Selection Map
- TimeSamples

Introduction

The **USD Crate** format is binary encoding of the USD scene graph. This document aims to document the layout of the this format.

> **Warning**
>
> This is a Work in Progress documentation of the USD Crate format. It is not guaranteed to be an exact description of the format right now. Contributions to improve accuracy are welcome.
>
> If areas are unspecified, or if a discrepancy occurs, the implementation in the official USD library should always take precedence as being the canonical implementation of the format.
>
> It is **NOT** yet recommended to make your own implementation based on this document , and we encourage developers to use the official implementation.
>
> This document does not describe the composition elements of USD. Implementing this document will only allow for reading a single USD layer.

Versions

This document only aims to document version *0.9.0* of the Crate format and higher. For older versions of the Crate format, please refer to the USD code implementations.

These older crate files are exceedingly rare outside of Pixar and very early USD adopters.

Version History

Even though the spec only starts with 0.9.0, it is useful to understand the high level history of versions

- 0.9.0: Added support for the timecode and timecode[] value types.
- 0.8.0: Added support for SdfPayloadListOp values and SdfPayload values with layer offsets.
- 0.7.0: Array sizes written as 64 bit ints.
- 0.6.0: Compressed (scalar) floating point arrays that are either all ints or can be represented efficiently with a lookup table.
- 0.5.0: Compressed (u)int & (u)int64 arrays, arrays no longer store '1' rank.
- 0.4.0: Compressed structural sections.
- 0.3.0: (broken, unused)
- 0.2.0: Added support for prepend and append fields of SdfListOp.
- 0.1.0: Fixed structure layout issue encountered in Windows port.
- 0.0.1: Initial release.

Order of Reads

The Crate format is designed for minimal parsing on file load.

To achieve this, the general structure of the file is set up in the preamble section below that must be read first.

Value types are parsed on demand from there on out.

Integers

Integers are stored with the least significant bit first to allow for fast loading on Little Endian systems.

Index

Many fields reference into an another section to get their value. USD uses unsigned 32 bit integers as an index unless otherwise specified.

Compression

LZ4 Compression

Various section and data blocks use LZ4 Compression.

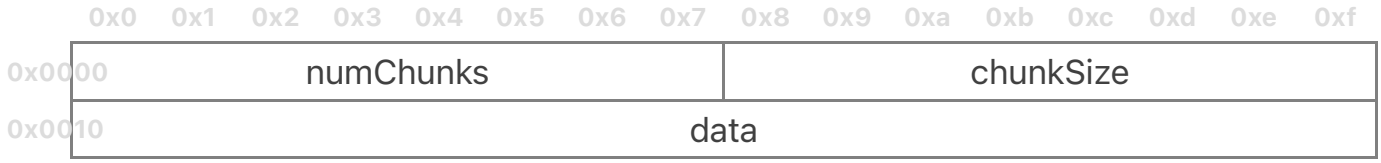
USD vendors the LZ4 library with modifications and can be found here. It is currently using version 1.9.2 of the library.

The first byte of any buffer passed to be decrypted stores the number of chunks used.

If the number of chunks is 0, then the buffer is decompressed as a whole.

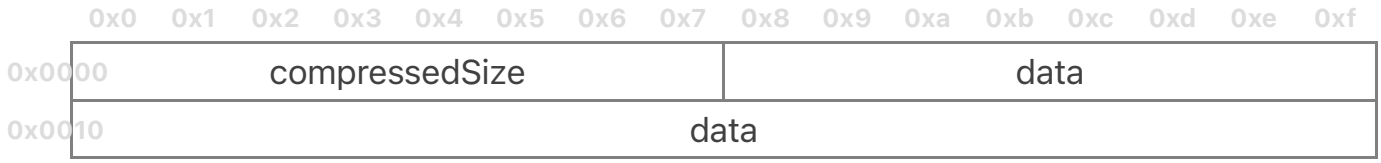
If the number of chunks is specified as higher than zero, then the first byte of each chunk represent the size of the chunk, followed by the chunk data which can be decompressed.

In the table below, chunkSize is not provided if numChunks is zero



Integer Compression

> Compressed integers are stored as a contiguous , homogenous array of either 32-bit or 64-bit integers.
>
> Once read, the array of integers can be decompressed using the LZ4 algorithm.



Preamble

The Crate format has a preamble that must be read prior to parsing the rest of the file.

This contains all the structural information that is required to identify Prim specifications and their resulting attributes.

Bootstrap

Every USD file starts with a Bootstrap that tells us information to parse the file.

Files must start with an identifier (PXR-USDC), the version number and the offset to the Table of Contents. There is additional reserved space at the end of the bootstrap for future use.

Version numbers are stored as Major , Minor and Patch followed by unused bytes.

	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xa	0xb	0xc	0xd	0xe	0xf
0x0000	identifier								version							
0x0010	toc_offset								reserved							

Table of Contents

The Table of Contents are found at the byte offset given by the bootstrap above.

It consist of named sections , as documented below.

Sections

Sections define a name as well as their start location and size in bytes. Section names are 16 characters wide

	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xa	0xb	0xc	0xd	0xe	0xf
0x0000	name															

0x0010	start	size

There are several sections with their own specific internal structures.

Sections should be parsed in the order below since many sections depend on their predecessors.

Tokens

The TOKENS section defines all the tokens within the file in their compressed form. The section must be null terminated.

It starts with an unsigned 64-bit integer representing the number of tokens. Following that are two unsigned 64-bit integers representing the uncompressed and compressed size respectively.

The data section follows the compressedSize and is that many bytes long. It must be uncompressed using TfFastCompression::DecompressFromBuffer which uses an LZ4 decompressor.

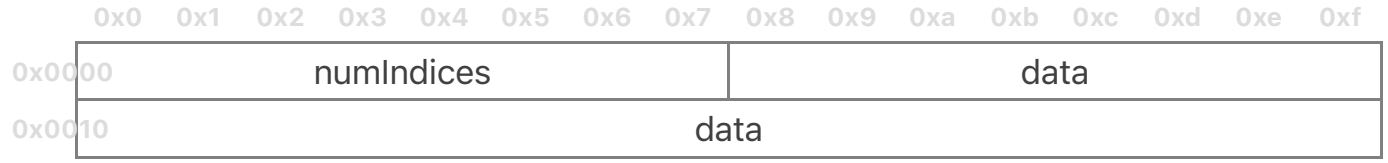
The uncompressed data section is a null delimited array of token strings.

	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xa	0xb	0xc	0xd	0xe	0xf
0x0000	numTokens								uncompressedSize							
0x0010	compressedSize								data							
0x0020	data															

Strings

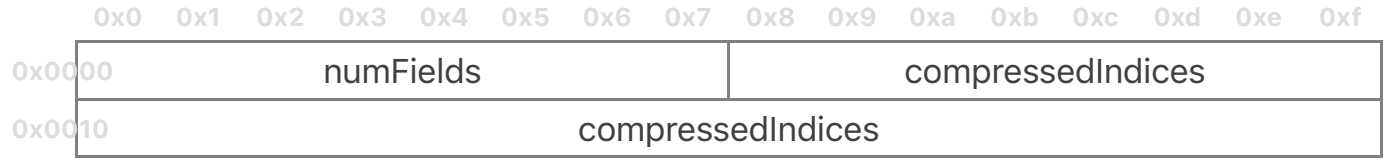
The STRINGS section is a vector of Indexes into the Tokens section.

It starts with an unsigned 64-bit integer representing the number of strings. Following this is a contiguous array of index values.

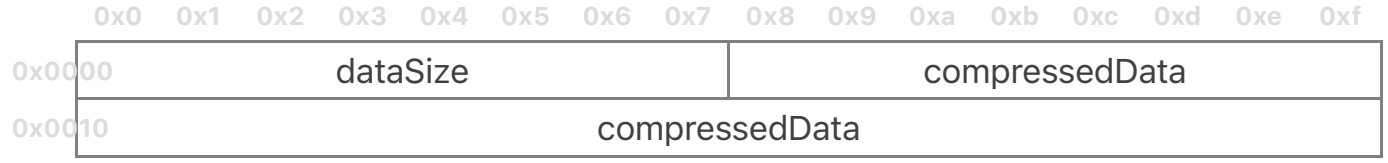


Fields

The FIELDS section stores the TfToken index and associated VtValue The indices are stored first as an LZ4 compressed set of Index values.



This is followed by an unsigned 64-bit integer representing the compressed size of the Value Representations. Following the size is the LZ4 compressed set of value representations.

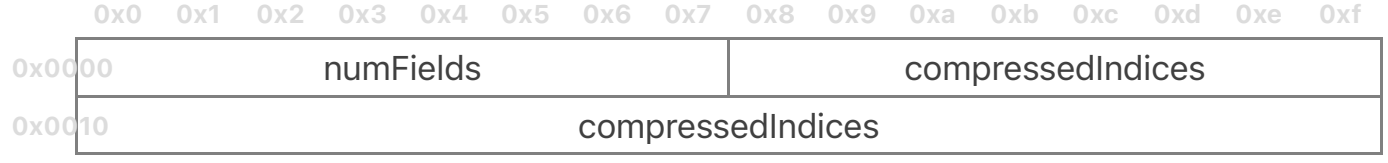


Field Sets

The FIELDSETS section stores a grouping of fields that are presented together.

The section starts with an unsigned 64-bit integer representing the number of indexes.

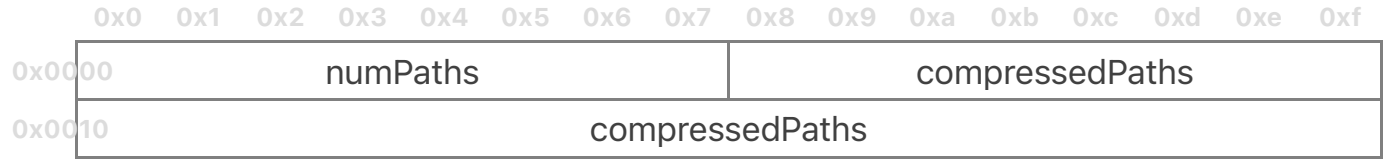
Field Sets are stored in a flat list of indexes into the FIELDS section, where groups are terminated by a default initialized FieldIndex (0).



Paths

The PATHS section stores a list of compressed SdfPaths used in the file. To start , the section starts with an unsinged 64-bit integer for the number of total paths.

See below for details on the path compression algorithm.



The compressed paths are stored as a series of indexed tokens that make up the path.

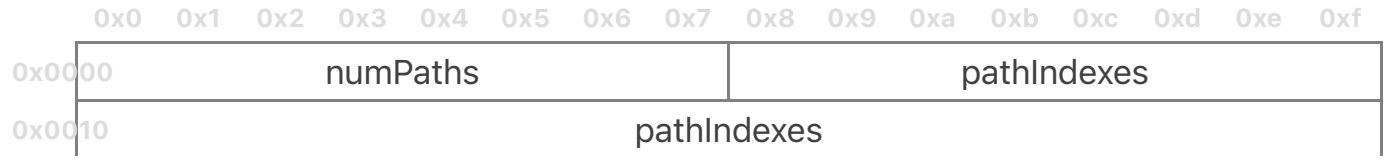
To start is a 64 bit unsigned integer that represents the number of paths.

Following this is a compressed integer array of Index's into the Tokens section that represent the path.

Next is the Element Token Index array. It consists of a compresed integer array of signed 32-bit integers. Positive elements represent an index to which path element should be added to the parent to construct the full path. Negative elements are prim property path elements.

Jumps define the hierarchy of this path. This optimizes storage for broad hierarchies over deep ones. They are stored as compressed array of signed 32-bit integers.

- 0 represents only a sibling.
- -1 represents only a child.
- -2 represents a leaf.
- Positive numbers define sibling offset relative to the current index.



0x0020	elementTokenIndexes
0x0030	jumps

Once you've read the paths section, you should 3 arrays that will let you build the paths:

- Path Indexes
- Element Token Indexes
- Jumps

To build the *path array* we need to recurse over the data. Start with the current index (X) being 1, to represent the first iteration. This will reflect the implicit root ("/") of the layer.

Once you have this element, add it to the path array at the index you find in the Path Index array at X. Use this path as your parent path for future iterations.

Now check the Jumps array at Index N to see if the path has siblings or children.

- It has children if the value is either greater than 0 or -1
- It has siblings if the value is greater or equal to 0

If it has siblings, you get the sibling index (S) by taking N and adding the value in the jumps array at X, then adding 1. Use the current element as the parent path and run this iteration again with the sibling index as X, and whatever parent path you've built thus far.

Building siblings is designed to be a parallelizable algorithm.

Once siblings are evaluated, or if the element had a child, you should then reset the parent path to whatever is stored in your Path array at the index from the Path Indexes array at X.

Repeat the process with an incrementing index (X++) as long as Jumps shows the element has children or siblings to process.

For any iterations (X), where a parent path has been calculated by a previous iteration (X-1), the new path is defined by looking up the Element Token Index array at X to get the element token index (E).

If the value of E is less than 0, it is a prim property path, otherwise it is another prim path.

Use the abs(value) of this index E to look up the token in your Tokens section at that index. Append this token to the parent path as either an element or a property to get your new path.

As before with the root iteration, add this to your Paths array at the index derived from looking up the Path Indexes array at your current iteration value (X).

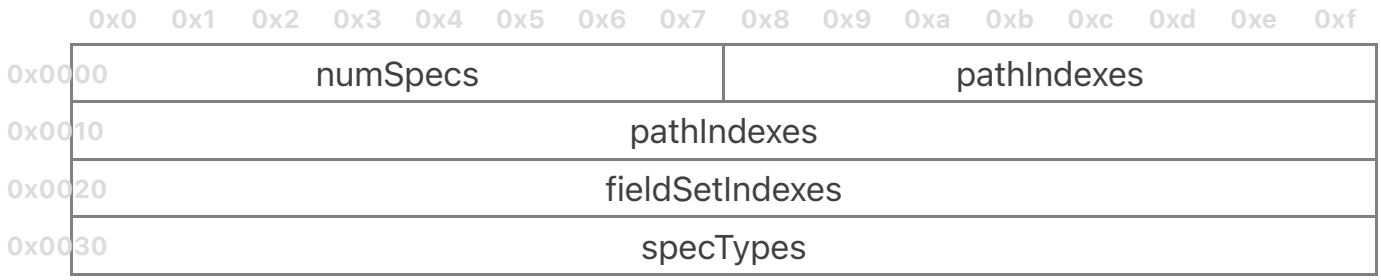
Specs

The SPECS section combines the data from the previous sections and creates a resulting PrimSpec.

The section starts with an unsigned 64-bit integer representing the number of specs.

The section is then made up of 3 compressed integer arrays:

- Path indexes consisting of Indexes into the Path section
- Field Set indexes consisting of Indexes into the Field Sets section.
- SpecTypes which are a series of unsigned 32-bit integers corresponding to the SpecType enum.



Spec Types are integer backed enums.

Certain types are marked as being **Internal to Pixar**. These are implementations of Pixar's internal software and are therefore reserved for use. Though they are documented below for completeness, they should not be used for other purposes.

Spec Type	Key	Description
Unknown	0	An unknown type
Attribute	1	Attributes under a Prim Spec
Connection	2	Connections between rel attributes
Expression	3	Scripted Expressions or named plugin expressions (Interna
Mapper	4	Used to modify the value flowing through a connection (Int
MapperArg	5	Arguments for a mapper (Internal to Pixar)
Prim	6	A Prim specifier
PseudoRoot	7	The Pseudo Root that exists for all Sdf Layers
Relationship	8	A relationship description
RelationshipTarget	9	The Relationship target
Variant	10	A definition of a specific variant
VariantSet	11	A group of variants]
NumSpecTypes	12	A sentinel to represent the number of types

Constructing the Layer

Once you have read the Preamble, you can then start constructing the prim hierarchy of this individual USD Crate file that can act as a layer within a USD composition.

Warning

> As noted above, implementation of this document only allows for reading a single USD file layer. It does not describe composition however as that is a higher level functionality of the USD library.

You should now have a mapping of:

- Prim and Property Paths from the Paths section
- Fields consisting of their associated Token and Value Representation.
- Field Sets consisting of groups of Fields
- Specs that associate paths with Field Sets and a Spec Type

Therefore each node consists of:

- A Path
- A SpecType
- A Field Set

Fields are made up of Value Representations that are deferred for parsing. See the section below on how to parse the range of Value Representations.

Layer Metadata

Layer Metadata is derived from the Field Sets on the PseudoRoot.

Common metadata in layers are, but not limited to:

Name	Type	Description
comment	string	Top level comment for this layer
customLayerData	dictionary	Custom user specified data
defaultPrim	token	The prim to use when this layer is referenced
documentation	string	Top level documentation for this layer
metersPerUnit	double	The scale unit for this layer

primChildren	token[]	A list of top level children to limit the layer to
timeCodesPerSecond	double	The time unit used
upAxis	token	The Up Axis of the scene

Prims

Prims have the spec type of Prim.

Prims are defined by the field set as below. However prims may have many more fields than this, especially depending on the schema type of the prim.

Name	Type	Description
specifier	SdfSpecifier	Whether this is a def, over or class
typeName	token	The optional prim Type
kind	token	The kind of this prim (component, assembly
active	bool	Whether this prim is active or not
apiSchems	TokenListOp	The API schemas to apply to this prim
variantSelection	VariantSelectionMap	The variants selected for this prim spec
references	ReferencesListOp	A list of references this prim should use in c
inherits	PathListOp	A list of prims this inherits from
properties	token[]	List of name of prim properties
primChildren	token[]	List of child prims to limit to
comment	string	The comment for this prim
documentation	string	The documentation for this prim

Variant Sets

Variant Sets have the VariantSet spec type, and the following fields

Name	Type	Description
variantChildren	token[]	A list of the names of the variants in this set

Variants

Variants have the Variant Spec Type. They have the following fields.

Variants encapsulate their own hierarchy for use in composition later.

Properties (Attribute)

Properties are of the Attribute SpecType. They are the individual data members of each composed prim.

The fields for properties are, but are not limited to:

Name	Type	Description
typeName	token	The name of the type
custom	bool	Whether or not this is a custom property
variability	variability	See the Variability section below
default	Value	The static value
timeSamples	TimeSamples	The TimeSampled animated values
connectionPaths	PathListOp	The list of paths this is connected to

Value Representations

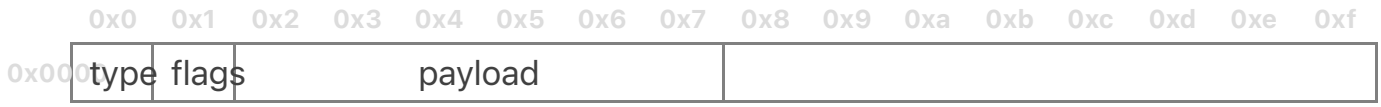
Crate stores Value Representations as data blobs that are read on demand. This allows large USD scenes to be read incredibly quickly by deferring all data reads to the latest point possible.

Value reps are stored as an unsigned 64 bit integer.

The first byte refers to the type enum value. The second byte has bit flags to represent characteristics of the type:

- Array Bit Mask is *1ull << 63* or *0x8000000000000000*
- Inlined Bit Mask is *1ull << 62* or *0x4000000000000000*
- Compressed Bit Mask is *1ull << 61* or *0x2000000000000000*

This is followed by 6 bytes of data.



If possible, we attempt to store certain values directly in the local data, such as ints, floats, enums, and special-case values of other types like zero vectors, identity matrices, etc...

For values that aren't stored inline (offset types), the 6 data bytes are the offset from the start of the file to the value's location.

Refer to the **Value Types** section below for documentation on different value types.

Inlined Types

Inlined types are stored directly in the payload of the value representation.

They cannot have the compressed or array bits set.

For types of a single dimension, you can simply cast the data bytes to the given type.

For single dimensioned types like **Vec2i** , elements are stored as signed 8 bit integers and cast to the native type of the container.

For multidimensional type like **Matrix2D**, the data is stored as the diagonal signed 8 bit integers and then cast to the native type of the type.

E.g a **Matrix3D** is stored as

```
1 – –
– 1 –
– – 1
```

Singular Offset Types

If a value representation has the array bit set to False, it can be treated as a singular value. The data of non-inlined types are stored at the offset defined by the payload value.

The bytes at the offset are a direct memory representation of each types' payload or are parsed based on the specifics of the types' implementation.

Array Offset Types

Array values may be stored in multiple ways.

If the payload of a Value Rep is 0, then the array can be assumed to be empty.

Uncompressed arrays use an unsigned 64 bit integer at the head of the data to represent the number of elements. Elements are stored as a contiguous array of the singular type used.

Array compression will vary based on the type of the value representation.

Compressed Integer Arrays

Compressed integer arrays are stored with the element count as an unsigned 64bit integer. Data can then be read with the standard integer compression algorithm.

Arrays that are 16 bytes or smaller are not compressed.

Compressed Floating Point Arrays

Compressed float arrays start with an unsigned 64 bit integer to represent their element count.

Arrays that are 16 bytes or smaller are not compressed.

Following this, a char is used to represent the array encoding scheme:

- **i** for integer compression (follow the compressed integer array logic)
- **t** for lookup tables (described below)

The size of the lookup table (LUT) is a 32bit unsigned integer. The LUT data is read as a contiguous array of the given type. Following that is a compressed array of integers representing the indexes that should be used to populate the output array by looking up the LUT indices in order.

Value Types

The following types are valid Value Representation types in USD.

Type Table

The following section enumerates the USD type table used to identify value types.

All type values are documented below the table.

Name	ID	C++ Type	Supports Array
Invalid	0		
Bool	1	bool	✓
UChar	2	uint8_t	✓
Int	3	int	✓
UInt	4	unsigned int	✓
Int64	5	int64_t	✓

UInt64	6	uint64_t	✓
Half	7	GfHalf	✓
Float	8	float	✓
Double	9	double	✓
String	10	std::string	✓
Token	11	TfToken	✓
AssetPath	12	SdfAssetPath	✓
Quatd	16	GfQuatd	✓
Quatf	17	GfQuatf	✓
Quath	18	GfQuath	✓
Vec2d	19	GfVec2d	✓
Vec2f	20	GfVec2f	✓
Vec2h	21	GfVec2h	✓
Vec2i	22	GfVec2i	✓
Vec3d	23	GfVec3d	✓
Vec3f	24	GfVec3f	✓
Vec3h	25	GfVec3h	✓
Vec3i	26	GfVec3i	✓
Vec4d	27	GfVec4d	✓
Vec4f	28	GfVec4f	✓
Vec4h	29	GfVec4h	✓
Vec4i	30	GfVec4i	✓
Matrix2d	13	GfMatrix2d	✓
Matrix3d	14	GfMatrix3d	✓
Matrix4d	15	GfMatrix4d	✓
Dictionary	31	VtDictionary	

TokenListOp	32	SdfTokenListOp	
StringListOp	33	SdfStringListOp	
PathListOp	34	SdfPathListOp	
ReferenceListOp	35	SdfReferenceListOp	
IntListOp	36	SdfIntListOp	
Int64ListOp	37	SdfInt64ListOp	
UIntListOp	38	SdfUIntListOp	
UInt64ListOp	39	SdfUInt64ListOp	
PathVector	40	SdfPathVector	
TokenVector	41	std::vector<TfToken>	
Specifier	42	SdfSpecifier	
Permission	43	SdfPermission	
Variability	44	SdfVariability	
VariantSelectionMap	45	SdfVariantSelectionMap	
TimeSamples	46	TimeSamples	
Payload	47	SdfPayload	
DoubleVector	48	std::vector<double>	
LayerOffsetVector	49	std::vector<SdfLayerOffset>	
StringVector	50	std::vector<std::string>	
ValueBlock	51	SdfValueBlock	
Value	52	VtValue	
UnregisteredValue	53	SdfUnregisteredValue	
UnregisteredValueListOp	54	SdfUnregisteredValueListOp	
PayloadListOp	55	SdfPayloadListOp	
TimeCode	56	SdfTimeCode	✓
NumTypes	57		

Base Types

The following types are foundational types that are used to compose other types.

Bool

The **Bool** type is a basic Binary boolean that can be checked by taking any non-zero number as True.

UChar

An unsigned character bit , represented by a single unsigned 8-bit integer.

Int

A signed 32-bit integer

Uint

An unsigned 32-bit integer

Int64

A signed 64-bit integer

UInt64

An unsigned 64-bit integer

Half

A 16-bit floating point data type

Float

A 32-bit floating point data type

Double

A 64-bit floating point data type

String

Strings are stored in the file as an index to a token in the Token section.

Token

Tokens are stored in the file as an index to a token in the Token section.

AssetPath

AssetPaths are stored in the file as an index to a token in the Token section.

Payload

A payload represents a prim reference to an external layer. It is similar to a reference but allow for deferred loading.

The first field is the asset layer path, represented by an index to an asset path in the Strings section. An empty string represents an internal reference.

This is followed by an index into the Paths section representing the prim to use. If no prim is specified, then use the defaultPrim or fallback to the first top level prim.

Finally, it is followed by 16-bytes that represent the layer offset.

ValueBlock

A special value type that can be used to explicitly author an opinion for an attribute's default value or time sample value that represents having no value. Note that this is different from not having a value authored.

Value

A Value type allows for an indirect pointer to another value somewhere else in the file.

It is represented by signed 64-bit integer offset which points to a Value Representation stored at the offset from the current seek pointer in the file.

It is important to guard against recursion here so that pointers don't create an infinite loop. The default behaviour is to return an empty value if a recursion is detected.

UnregisteredValue

A representation of unregistered metadata field values.

Read in the same way as a Value, as an indirection pointer to the real data. Possible value types are * Strings * Dictionaries * UnregisteredValueListOps

If the type is something else, return an empty value.

TimeCode

A single Double that represents an SdfTimeCode

Other Types

When traversing a USD crate file, other data types used by SDF are involved, and described here. These are not directly referencable as Value Representations but are used by other container types below.

References

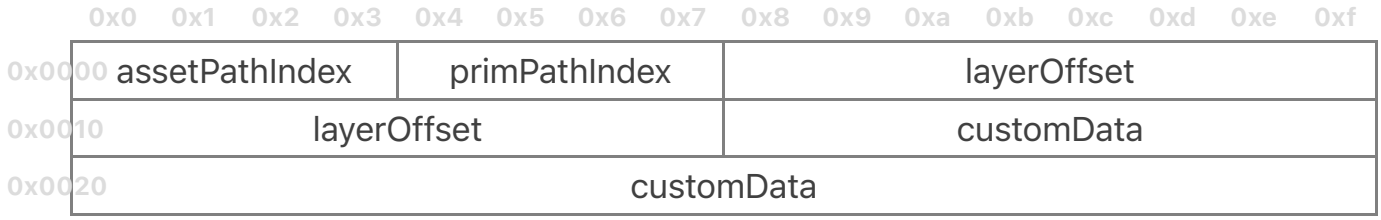
A reference type represents an SdfReference and all its metadata.

The first field is an index into the strings section which represents the asset path for the reference. An empty string represents an internal reference.

This is followed by an index into the Paths section for the path of the Prim to reference. If the PrimPath is empty, the recomendation is to use the defaultPrim, falling back on the first top level prim.

Next is a layer offset value corresponding to an SdfLayerOffset represented by 16 bytes.

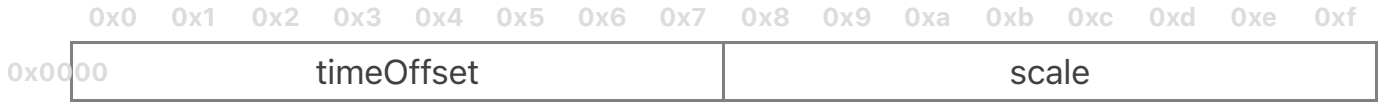
Finally, it's followed by a data blob representing a Dictionary.



Layer Offset

A layer offset represents a time offset and scale between layers. It is represented by two doubles.

The first represents the time offset to be used , and the second the scale. Scale should always be applied before the offset.



Quaternions

Quaternions are represented by four contiguous elements of a given type.

The first three make up the imaginary coefficient. The last the element is the real coefficient.

QuatD

A Quaternion using doubles as the core type.

QuatF

A Quaternion using floats as the core type.

QuatH

A Quaternion using halfs as the core type.

Vectors (Mathematical)

Vectors are fixed length contiguous arrays of a given type.

Vec2d

A vector with 2 doubles.

Vec2f

A vector with 2 floats.

Vec2h

A vector with 2 halfs.

Vec2i

A vector with 2 32-bit integers.

Vec3d

A vector with 3 doubles.

Vec3f

A vector with 2 floats.

Vec3h

A vector with 3 halfs.

Vec3i

A vector with 3 32-bit integers.

Vec4d

A vector with 4 doubles.

Vec4f

A vector with 4 floats.

Vec4h

A vector with 4 halfs.

Vec4i

A vector with 4 32-bit integers.

Matrix

Matrix types are NxN dimensional groups of Doubles. They are defined in a contiguous row-major order so *matrix[i][j]* refers to row **i** and column **j**.

Identity Matrix values have all cells zeroed out, except for the diagonal from top left (0,0) to bottom right (N,N) which are set to 1.

Matrix2d

A 2x2 matrix

Matrix3d

A 3x3 matrix

Matrix4d

A 4x4 matrix

Dictionary

A Dictionary is a key:value map of data.

The Value Representation starts with an unsigned 64-bit integer representing the number of elements in the dictionary.

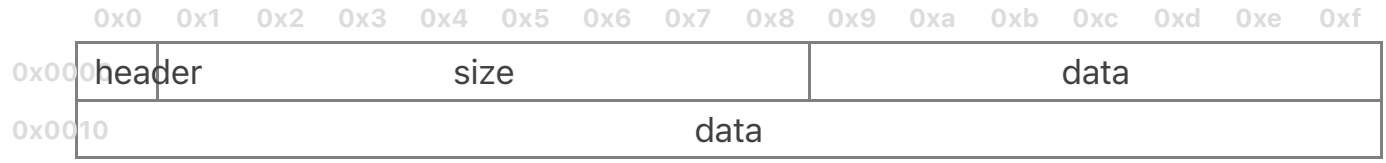
Keys are strings, stored as an index to a token in the Token section.

Values are a 64-bit unsigned integer representing the offset from the start of the file to a value representation.

List Operations

List Operations (ListOps) are a value type representing an operation that edits a list. It can make modifications like:

- Adding or removing items
- Reordering Items
- Replacing items



Header

ListOps have an 8-byte bitmasked header that determine what type of operation is being run.

In order, from least to most significant bit:

- **IsExplicit** : Removes all items and changes the list to be explicit
- **HasExcplicitItems** : Fills the list with the items included in this ListOp
- **HasAddedItems** : Adds the items included in this ListOp
- **HasDeletedItems** : Removes the items specified in this ListOp
- **HasOrderedItems** : Reorder the list based on the item order in this ListOp
- **HasPrependedItems** : Prepend the items from this ListOp
- **HasAppendedItems** : Append items from this ListOp

Contents

The header byte is then followed by an unsigned 64-bit integer representing the number of elements stored in the ListOp

Following this is a contiguous array of uncompressed elements of the given ListOp type.

TokenListOp

A ListOp made of Tokens, stored as an array of Indexes referencing the Tokens section.

StringListOp

A ListOp made of Strings, stored as an array of Indexes referencing the Tokens section.

ReferenceListOp

A ListOp consisting of References. See the References section below.

IntListOp

A ListOp consisting of signed 32-bit Integers

Int64ListOp

A ListOp consisting of signed 64-bit Integers

UIntListOp

A ListOp consisting of unsigned 32-bit Integers

UInt64ListOp

A ListOp consisting of unsigned 64-bit Integers

PayloadListOp

A ListOp consisting of a Payload. See the Payload section above.

UnregisteredValueListOp

A ListOp consisting of Unregistered Values. See the UnregisteredValue section above.

Vectors (Arrays)

Vectors are arrays of a given type stored contiguously. These are different in intent from the mathematical vectors listed above, and their naming reflects the container type in programming languages like C++.

They always start with a 64-bit unsigned integer reflecting the number of elements, which is followed by a contiguous array of the specific data type.

PathVector

Consists of Index signed integers referencing the Paths section

TokenVector

Consists of Index signed integers referencing the Tokens section

DoubleVector

Consists of Doubles

LayerOffsetVector

Consists of Layer Offsets

StringVector

Consists of Index signed integers referencing the Tokens section

Specifier

A 32-bit integer backed enum that defines the possible specifiers for a prim. These are representations of SdfSpecifier.

Spec Type	Key	Description
Def	0	Defines a concrete prim
Over	1	Overrides an existing prim

Class	2	Defines an abstract prim
NumSpecifiers	3	The number of possible specifiers

Permission

A 32-bit integer backed enum that defines permissions. Permissions control which layers may refer to or express opinions about a prim.

These are representations of SdfPermission.

Spec Type	Key	Description
Public	0	Public prims can be referred to by anything
Private	1	Private prims can only be referred within the local layer stacl
NumPermissions	2	The number of possible permissions

Variability

A 32-bit integer backed enum that defines variability for an attribute. Variability indicates whether an attribute is uniform or time varying.

These are representations of SdfVariability.

Spec Type	Key	Description
Varying	0	Can be animated
Uniform	1	Can only be static
NumVariability	2	The number of possible variability types

Variant Selection Map

A string:string map of reference variant set names to variants in those sets.

It starts with an unsigned 64-bit integer representing the number of elements.

The following data block is filled with pairs of Indexes into the String section stored as contiguous key:value pairs.

TimeSamples

Timesamples store a series of time varying ValueReps

It starts with a signed 64-bit integer that presents an offset from the current seek pointer in the file. This points to the location where time values are stored.

At this offset you'll find a Value Representation that consists of an array of the time values.

Following the original offset will be another signed 64-bit integer that represents an offset from its current seek position in the file.

This offset will point to a block the values for the time samples are stored. The head of this block is an unsigned 64-bit integer representing the number of Value Representations that will be found after it.

These values are index-mapped to the time samples that were read.