Basic Data Encoding

On this page:

Encoding for Sizes

Many of the types involved in the data encoding, as well as several protocol message components, have an associated size or count. A size is a non-negative number. Sizes and counts are encoded in one of two ways:

1. If the number of elements is less than 255, the size is encoded as a single `byte` indicating the number of elements.
2. If the number of elements is greater than or equal to 255, the size is encoded as a `byte` with value `255`, followed by an `int` indicating the number of elements.

Using this encoding to indicate sizes is significantly cheaper than always using an `int` to store the size, especially when marshaling sequences of short strings: counts of up to 254 require only a single byte instead of four. This comes at the expense of counts greater than 254, which require five bytes instead of four. However, for sequences or strings of length greater than 254, the extra byte is insignificant.

Encoding for Encapsulations

An encapsulation is used to contain variable-length data that an intermediate receiver may not be able to decode, but that the receiver can forward to another recipient for eventual decoding. An encapsulation is encoded as if it were the following structure:

Slice

```
struct Encapsulation {
    int size;
    byte major;
    byte minor;
    // [... size - 6 bytes ...]
};
```

The `size` member specifies the size of the encapsulation in bytes (including the `size`, `major`, and `minor` fields). The `major` and `minor` fields specify the encoding version of the data contained in the encapsulation. The version information is followed by `size-6` bytes of encoded data.

All the data in an encapsulation is context-free, that is, nothing inside an encapsulation can refer to anything outside the encapsulation. This property allows encapsulations to be forwarded among address spaces as a blob of data.

Encapsulations can be nested, that is, contain other encapsulations.

An encapsulation can be empty, in which case the value of `size` is 6.

Encoding for Slices

The encoding format of slices changed in version 1.1.

Slice Encoding version 1.0

Exceptions and classes may be subject to *slicing* if the receiver of a value only partially understands the received value (that is, only has knowledge of a base type, but not of the actual run-time derived type). To allow the receiver of an exception or class to ignore those parts of a value that it does not understand, exception and class values are marshaled as a sequence of slices (one slice for each level of the inheritance hierarchy). A slice is a byte count encoded as a fixed-length four-byte integer, followed by the data for the slice. (The byte count includes the four bytes occupied by the count itself, so an empty slice has a byte count of four and no data.) The receiver of a value can skip over a slice by reading the byte count *b*, and then discarding the next *b-4* bytes in the input stream.

Slice Encoding version 1.1

Version 1.1 of the encoding still marshals exceptions and classes as slices in conceptually the same manner as for version 1.0, but bit flags in the leading byte of each slice determine its format and content.

Type ID

The initial slice of a class or exception, representing the instance's most-derived type, always includes a type ID. For an exception, the type ID in the initial slice is encoded as a string. For a class, the type ID in the initial slice can either be encoded as a string, an index (if the same type ID has already been encoded in the current encapsulation), or a compact ID.

Whether any subsequent slices include some form of type ID depends on the format with which the value was encoded: to facilitate slicing an instance to a less-derived type, the sliced format includes a type ID in every slice, whereas the compact format excludes type IDs in subsequent slices to conserve space while sacrificing the slicing feature.

Optional Data Members

This flag is true if the slice includes any optional data members, which are encoded after all required members. If a slice encodes its size, the size includes the optional data members.

Object Indirection Table

This flag can only be true when using the sliced format. In this case, data members that refer to class instances are encoded as indices into an indirection table that immediately follows the slice. The slice's size does *not* include the indirection table. This flag should only be set to true when there is at least one non-nil object reference in the slice, that is, when the indirection table is not empty.

Slice Size

If this flag is true, a byte count encoded as a fixed-length four-byte integer immediately follows the type ID. (The byte count includes the four bytes occupied by the count itself, so an empty slice has a byte count of four and no data.) The byte count indicates the number of bytes occupied by the encoding for the required and optional data members, but does not include the size of an object indirection table, if present.

A receiver can skip over a slice by reading the byte count *b*, and then discarding the next *b-4* bytes in the input stream. If an object indirection table is present, the receiver must then decode the table.

If this flag is false, it implies that the sender used the compact format and therefore skipping slices is not possible. The receiver must know the most-derived type in this situation otherwise decoding will fail.

Last Slice

This flag indicates whether the current slice is the last slice of the instance.

Slice Flags

The table below shows how to interpret the bit flags in the leading byte of a slice:

Bit number	Description
0-1	0 = no type ID is encoded for the slice
	1 = type ID is encoded as a string
	2 = type ID is an index encoded as a size
	3 = type ID is a compact ID encoded as a size
2	Whether or not the slice contains optional data members
3	Whether or not the slice contains an indirection table
4	Whether or not the slice size follows the type ID
5	If 0, more slices will follow, if 1, this is the last slice
6	Reserved for future use

7	Reserved for future use

Bit flags for a slice.

Encoding for Basic Types

The basic types are encoded as shown in the table. Integer types (`short`, `int`, `long`) are represented as two's complement numbers, and floating point types (`float`, `double`) use the IEEE standard formats [1]. All numeric types use a little-endian byte order.

Type	Encoding
`bool`	A single byte with value `1` for `true`, `0` for `false`
`byte`	An uninterpreted byte
`short`	Two bytes (LSB, MSB)
`int`	Four bytes (LSB .. MSB)
`long`	Eight bytes (LSB .. MSB)
`float`	Four bytes (23-bit fractional mantissa, 8-bit exponent, sign bit)
`double`	Eight bytes (52-bit fractional mantissa, 11-bit exponent, sign bit)

Encoding for basic types.

Encoding for Strings

Strings are encoded as a size, followed by the string contents in UTF-8 format [2]. Strings are not NUL-terminated. An empty string is encoded with a size of zero.

Encoding for Sequences

Sequences are encoded as a size representing the number of elements in the sequence, followed by the elements encoded as specified for their type.

Encoding for Dictionaries

Dictionaries are encoded as a size representing the number of key-value pairs in the dictionary, followed by the pairs. Each key-value pair is encoded as if it were a `struct` containing the key and value as members, in that order.

Encoding for Enumerators

The encoding format of enumerators changed in version 1.1.

Enumerator Encoding version 1.0

The number of bytes required to encode an enumerator in version 1.0 is determined by the largest value in the enumeration. In enumerations with no custom enumerator values, the largest value is the number of enumerators less one; the value encoded for an enumerator is its ordinal value in the definition, with the first enumerator having the value zero. For example, the largest value in the following enumeration is 2:

Slice

```
enum Fruit { Apple, Pear, Orange }; // Encoded values: Apple = 0, Pear = 1, Orange = 2
```

When an enumeration includes custom values, the encoding uses the enumerator's assigned value, which is not necessarily the same as its ordinal value. Consider this example:

Slice

```
enum Fruit { Apple = 1, Pear = 3, Orange }; // Encoded values: Apple = 1, Pear = 3, Orange = 4
```

The largest value in this case is 4.

An enumerator is encoded as follows:

- If the largest value is in the range 0..126, the enumerator's value is marshaled as a `byte`.
- If the largest value is in the range 127..32766, the enumerator's value is marshaled as a `short`.
- If the largest value is greater than 32766, the enumerator's value is marshaled as an `int`.

> Changing the definition of an enumeration can break compatibility with existing applications. For example, if enumerators are added, removed, or changed such that the largest value crosses one of the thresholds shown above, the encoded form of the enumerators will change and cause marshaling errors unless you rebuild all applications that use this definition.

Enumerator Encoding version 1.1

An enumerator is encoded as a size, meaning the encoding of an enumerator requires one byte if its value is less than 255, or five bytes if its value is 255 or greater. The encoding uses the Slice value of the enumerator, which is not necessarily the same as its ordinal value. Repeating the previous example:

Slice

```
enum Fruit { Apple = 1, Pear = 3, Orange }; // Encoded values: Apple = 1, Pear = 3, Orange = 4
```

Although enumerator `Pear` may have ordinal value 1 in some language mappings (notably, Java), the encoding uses its Slice value of 3.

Encoding for Structures

The members of a structure are encoded in the order they appear in the `struct` declaration, as specified for their types.

See Also

- Protocol Messages
- Protocol and Encoding Versions
- Data Encoding for Exceptions
- Data Encoding for Classes

References

1. Institute of Electrical and Electronics Engineers. 1985. *IEEE 754-1985 Standard for Binary Floating-Point Arithmetic*. Piscataway, NJ: Institute of Electrical and Electronic Engineers.
2. Unicode Consortium, ed. 2000. *The Unicode Standard, Version 3.0*. Reading, MA: Addison-Wesley.