Data Encoding for Optional Values

Slice definitions can use optional values as parameters and as data members. To make optional values truly useful, the receiver needs the ability to gracefully ignore an optional value that it does not recognize. Although the Ice encoding omits type information and therefore is not self-describing, the encoding for optional values must include enough information for a receiver to determine how many bytes an optional value occupies so that it can skip to the next value in the stream. Despite this requirement, the encoding rules minimize the overhead associated with optional values, as you will see below.

On this page:

Overview of the Optional Value Encoding

The encoding for optional parameters and data members follows these general rules:

- Each optional value has a corresponding integer tag that uniquely identifies it.
- Optional values always appear *after* any required values.
- Optional values must be sorted by their tags. (Unlike required values, order of declaration does not affect optional values.)
- An optional value is encoded only if the sender has supplied a value.

Optional data members of a class or exception appear in the slice after any required data members and before the indirection table, if present. The slice flags must indicate the presence of optional data members, which are included in the byte count for the slice. If a slice contains optional data members, the byte value `255` must be written after the last optional data member; this marker denotes the end of the optional data members and is also included in the byte count for the slice.

The encoding does not use an explicit end marker for optional parameters; the end of the encapsulation also marks the end of any optional parameters.

An optional value is encoded as the tuple *<type, tag, value>*, where *type* is the *optional type* that tells the receiver how to determine the number of bytes occupied by the value. The value itself is marshaled using the standard Ice encoding rules for its Slice type.

Optional values require encoding version 1.1.

Encoding for Optional Types and Tags

The first byte of an encoded optional value includes the optional type, and may also include the tag. The optional type occupies the first three bits of this byte, as described in the table below:

Name	Value	Description	Used for Slice type
F1	0	The value is encoded in one byte.	bool, byte
F2	1	The value is encoded in two bytes.	short
F4	2	The value is encoded in four bytes.	int, float
F8	3	The value is encoded in eight bytes.	double, long
Size	4	The value is encoded as a size.	enum
VSize	5	A leading size value indicates the number of bytes occupied by the value.	string, fixed-size structure, container of fixed-size elements
FSize	6	A leading 32-bit integer indicates the number of bytes occupied by the value.	variable-size structure, container of variable-size elements
Class	7	A class reference or inline instance.	class

The next five bits of the leading byte contain the tag, but only if the tag value is less than 30. Otherwise, the next five bits contain the value 30 as a marker to indicate that the tag value is encoded as a size starting with the next byte. As you can see, using tag values in the range 0 to 29 produces the most compact encoding.

Variable-size types whose encoded size cannot be determined in advance use the FSize optional type, where "FSize" denotes a leading fixed-length (32-bit) size. For these types, the sender reserves four bytes to hold the size, encodes the value as usual, then replaces the four bytes with the actual encoded size. This strategy avoids the need to shift the encoded data in the buffer, at the expense of potentially consuming more bytes than necessary to encode the size.

Fixed-size types use the VSize optional type, where "VSize" denotes a leading variable-length size. The sender can determine the encoded size of these types in advance, and therefore encodes it as a size followed by the value as usual.

Strings also use the VSize optional type but do not require an additional size because the string encoding already includes a leading size. The same is true for sequences of `bool` and `byte`.

The optional type Class represents a class reference or inline instance. Note that a receiver must decode an inline instance even if it does not recognize the tag value because the instance may be referenced by other parameters.

The following table describes the encoding of Slice types:

Slice type	Optional type	Data encoding	Notes
`bool`, `byte`	F1	value	
`short`	F2	value	
`int`, `float`	F4	value	
`long`, `double`	F8	value	
Proxy	FSize	int + data	32-bit integer holds the size of the encoded proxy
`class`	Class	reference or instance	
`enum`	Size	size	Enumerator encoded as a size
`string`	VSize	value	The encoded data for the string already contains a leading size
`sequence<bool>`, `sequence<byte>`	VSize	value	The encoded data for the sequence already contains a leading size
`sequence<`*fixed-size type*`>`, `dictionary<`*fixed-size key*`,`*fixed-size value*`>`	VSize	size + value	Size can be computed before encoding the container
`sequence<`*variable size type*`>`, `dictionary<`*variable-size key*`,`*variable-size value*`>`	FSize	int + value	32-bit integer holds the size of the container

Examples of Optional Value Encoding

The examples presented below demonstrate the encoding for optional values with typical use cases.

Optional Parameters Example

The following Slice operation makes use of optional input and output parameters:

Slice

```
bool op1(byte b,
         optional(2) string name,
         short sh,
         optional(1) long count,
         out double d,
         out optional(300) Object* p);
```

Suppose the parameters have the values shown in the table below:

Member	Type	Value	Marshaled size (in bytes)
`b`	`byte`	`77`	1
`name`	`string`	`"joe"`	4
`sh`	`short`	`99`	2

`count`	`long`	`88`	8
`d`	`double`	`3.14`	8
`p`	`proxy`	nil	2
return value	`bool`	`true`	1

The parameters of the outgoing request are encoded in an encapsulation with all required parameters first, in order of declaration, followed by the optional parameters sorted by tag:

Marshaled value	Size in bytes	Type	Byte offset
`77` (*b*)	1	`byte`	0
`99` (*sh*)	2	`short`	1
`11` (*optional type F8 + tag 1*)	1	`byte`	3
`88` (*count*)	8	`long`	4
`21` (*optional type VSize + tag 2*)	1	`byte`	12
`"joe"` (*name*)	4	`long`	13

Using tag values less than 30 allows the encoding to combine the optional type and the tag into the same byte using the expression `(Tag << 3) + Type`.

Now consider the contents of the reply message:

Marshaled value	Size in bytes	Type	Byte offset
`3.14` (*d*)	8	`double`	0
`true` (*return value*)	1	`bool`	8
`246` (*optional type FSize + marker 30*)	1	`byte`	9
`300` (*tag*)	5	`size`	10
2 (*32-bit FSize*)	4	`int`	15
nil (*p*)	2	`Object*`	19

The body of the reply message contains the out parameter `d`, the return value, and the optional parameter `p`. The tag value 300 is too large to combine with the optional type, therefore it appears immediately following the optional type encoded as a size. A proxy value uses the FSize optional type, meaning a 32-bit integer precedes the encoded value to specify its size.

Although the return value is required in this example, an optional return value is treated as if it were an optional out parameter.

The server here supplies a nil value for the optional proxy parameter. The client must not interpret this to mean that the optional parameter is unset; rather, the parameter is set, it just happens to be set to a nil value. If the server had supplied no value for the parameter, it would not appear in the encoding at all.

As an example, if the client does not recognize the optional proxy parameter, it can skip the parameter as follows:

1. Extract the byte containing the optional type
2. Examine the first three bits to determine the type
3. Examine the five remaining bits to determine the tag's status
4. A value of 30 means the tag is encoded separately
5. Extract the tag as a size
6. The optional type FSize means a 32-bit integer holds the value's size; extract the integer
7. Skip ahead the specified number of bytes
8. If we have reached the end of the encapsulation, there are no more optional parameters remaining

Optional Data Members Example

The only difference between the encoding for optional parameters and the encoding for optional data members is the latter uses a marker byte to signify the end of optional parameters in a slice. Consider the following definitions:

Slice

```
struct Color
{
    short red;
    short green;
    short blue;
};

class Shape
{
    optional(1) string label;
};

class Rectangle extends Shape
{
    int width;
    int height;
    optional(10) Color fill;
    optional(9) Color border;
    optional(11) float scale;
};
```

Suppose we are encoding an instance of `Rectangle` with the member values shown in the table below:

Member	Type	Value	Marshaled size (in bytes)
`label`	`string`	`"r1"`	3
`width`	`int`	`41`	4
`height`	`int`	`16`	4
`fill`	`Color`	`0,0,0`	6
`border`	`Color`	`255,255,255`	6
`scale`	`float`	`2.0`	4

Using the sliced format, the instance data looks as follows:

Marshaled value	Size in bytes	Type	Byte offset
`1` (*instance marker*)	1	`byte`	0
`21` (*slice flags: string type ID, size is present, optional members are present*)	1	`byte`	1
`"::Rectangle"` (*type ID*)	12	`string`	2
`34` (*byte count for slice*)	4	`int`	14
`41` (*`width`*)	4	`int`	18
`16` (*`height`*)	4	`int`	22
`77` (*optional type VSize + tag 9*)	1	`byte`	26
`6` (*VSize*)	1	`size`	27
`{0,0,0}` (*`border`*)	6	`Color`	28
`85` (*optional type VSize + tag 10*)	1	`byte`	34
`6` (*VSize*)	1	`size`	35
`{255,255,255}` (*`fill`*)	6	`Color`	36
`90` (*optional type F4 + tag 11*)	1	`byte`	42
`2.0` (*`scale`*)	4	`float`	43
`255` (*end marker*)	1	`byte`	47

53 (*slice flags: string type ID, size is present, optional members are present, last slice*)	1	`byte`	48
`"::Shape"` (*type ID*)	8	`string`	49
9 (*byte count for slice*)	4	`int`	57
13 (*optional type VSize + tag 1*)	1	`byte`	61
`"r1"` (*label*)	3	`string`	62
255 (*end marker*)	1	`byte`	65

Notice the use of an end marker (byte value 255) denoting the end of the optional data members in each slice.

See Also

- Optional Values
- Basic Data Encoding
- Data Encoding for Classes