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

Exceptions and classes are 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.

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

Enumerated values are encoded depending on the number of enumerators:

- If the enumeration has 1 - 127 enumerators, the value is marshaled as a `byte`.
- If the enumeration has 128 - 32767 members, the value is marshaled as a `short`.
- If the enumeration has more than 32767 members, the value is marshaled as an `int`.

The value is the ordinal value of the corresponding enumerator, with the first enumerator value encoded as zero.

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.