NAME

Sereal::Encoder – Fast, compact, powerful binary serialization

SYNOPSIS

```
use Sereal::Encoder qw(encode_sereal sereal_encode_with_object);

my $encoder = Sereal::Encoder->new({...options...});
my $out = $encoder->encode($structure);

# alternatively the functional interface:
$out = sereal_encode_with_object($encoder, $structure);

# much slower functional interface with no persistent objects:
$out = encode_sereal($structure, {... options ...});
```

DESCRIPTION

This library implements an efficient, compact-output, and feature-rich serializer using a binary protocol called *Sereal*. Its sister module Sereal::Decoder implements a decoder for this format. The two are released separately to allow for independent and safer upgrading. If you care greatly about performance, consider reading the Sereal::Performance documentation after finishing this document.

The Sereal protocol version emitted by this encoder implementation is currently protocol version 4 by default.

The protocol specification and many other bits of documentation can be found in the github repository. Right now, the specification is at <https://github.com/Sereal/Sereal/blob/master/sereal_spec.pod>, there is a discussion of the design objectives in <https://github.com/Sereal/Sereal/blob/master/README.pod>, and the output of our benchmarks can be seen at <https://github.com/Sereal/Sereal/wiki/Sereal-Comparison-Graphs>. For more information on getting the best performance out of Sereal, have a look at the ''PERFORMANCE'' section below.

CLASS METHODS

new

Constructor. Optionally takes a hash reference as first parameter. This hash reference may contain any number of options that influence the behaviour of the encoder.

Currently, the following options are recognized, none of them are on by default.

compress

If this option provided and true, compression of the document body is enabled. As of Sereal version 4, three different compression techniques are supported and can be enabled by setting `compress` to the respective named constants (exportable from the `Sereal::Encoder` module): Snappy (named constant: `SRL_SNAPPY`), Zlib (`SRL_ZLIB`) and Zstd (`SRL_ZSTD`). For your convenience, there is also a `SRL_UNCOMPRESSED` constant.

If this option is set, then the Snappy-related options below are ignored. They are otherwise recognized for compatibility only.

compress_threshold

The size threshold (in bytes) of the uncompressed output below which compression is not even attempted even if enabled. Defaults to one kilobyte (1024 bytes). Set this to 0 and `compress` to a non-`SRL_UNCOMPRESSED` value to always attempt to compress. Note that the document will not be compressed if the resulting size will be bigger than the original size (even if `compress_threshold` is 0).

compress_level

If Zlib or Zstd compressions are used, then this option will set a compression level: Zlib uses range from 1 (fastest) to 9 (best). Defaults to 6. Zstd uses range from 1 (fastest) to 22 (best). Default is 3.

snappy

See also the `compress` option. This option is provided only for compatibility with Sereal V1.

If set, the main payload of the Sereal document will be compressed using Google's Snappy algorithm. This can yield anywhere from no effect to significant savings on output size at rather low run time cost. If in doubt, test with your data whether this helps or not.

The decoder (version 0.04 and up) will know how to handle Snappy-compressed Sereal documents transparently.

Note: The `snappy_incr` and `snappy` options are identical in Sereal protocol v2 and up (so by default). If using an older protocol version (see `protocol_version` and `use_protocol_v1` options below) to emit Sereal V1 documents, this emits non-incrementally decodable documents. See `snappy_incr` in those cases.

snappy_incr

See also the `compress` option. This option is provided only for compatibility with Sereal V1.

Same as the `snappy` option for default operation (that is in Sereal v2 or up).

In Sereal V1, enables a version of the Snappy protocol which is suitable for incremental parsing of packets. See also the `snappy` option above for more details.

snappy_threshold

See also the `compress` option. This option is provided only for compatibility with Sereal V1.

This option is a synonym for the `compress_threshold` option, but only if Snappy compression is enabled.

croak_on_bless

If this option is set, then the encoder will refuse to serialize blessed references and throw an exception instead.

This can be important because blessed references can mean executing a destructor on a remote system or generally executing code based on data.

See also `no_bless_objects` to skip the blessing of objects. When both flags are set, `croak_on_bless` has a higher precedence then `no_bless_objects`.

freeze_callbacks

This option was introduced in Sereal v2 and needs a Sereal v2 decoder.

If this option is set, the encoder will check for and possibly invoke the `FREEZE` method on any object in the input data. An object that was serialized using its `FREEZE` method will have its corresponding `THAW` class method called during deserialization. The exact semantics are documented below under "FREEZE/THAW CALLBACK MECHANISM".

Beware that using this functionality means a significant slowdown for object serialization. Even when serializing objects without a `FREEZE` method, the additional method look up will cost a small amount of runtime. Yes, `Sereal::Encoder` is so fast that this may make a difference.

no_bless_objects

If this option is set, then the encoder will serialize blessed references without the bless information and provide plain data structures instead.

See also the `croak_on_bless` option above for more details.

undef_unknown

If set, unknown/unsupported data structures will be encoded as `undef` instead of throwing an exception.

Mutually exclusive with `stringify_unknown`. See also `warn_unknown` below.

stringify_unknown

If set, unknown/unsupported data structures will be stringified and encoded as that string instead of

throwing an exception. The stringification may cause a warning to be emitted by perl.

Mutually exclusive with `undef_unknown`. See also `warn_unknown` below.

warn_unknown

Only has an effect if `undef_unknown` or `stringify_unknown` are enabled.

If set to a positive integer, any unknown/unsupported data structure encountered will emit a warning. If set to a negative integer, it will warn for unsupported data structures just the same as for a positive value with one exception: For blessed, unsupported items that have string overloading, we silently stringify without warning.

max_recursion_depth

`Sereal::Encoder` is recursive. If you pass it a Perl data structure that is deeply nested, it will eventually exhaust the C stack. Therefore, there is a limit on the depth of recursion that is accepted. It defaults to 10000 nested calls. You may choose to override this value with the `max_recursion_depth` option. Beware that setting it too high can cause hard crashes, so only do that if you **KNOW** that it is safe to do so.

Do note that the setting is somewhat approximate. Setting it to 10000 may break at somewhere between 9997 and 10003 nested structures depending on their types.

canonical

Enable all options which are related to producing canonical output, so that two strucutures with similar contents produce the same serialized form.

See the caveats elsewhere in this document about producing canonical output.

Currently sets the default for the following parameters: `canonical_refs` and `sort_keys`. If the option is explicitly set then this setting is ignored. More options may be added in the future.

You are warned that use of this option may incur additional performance penalties in a future release by enabling other options than those listed here.

canonical_refs

Normally `Sereal::Encoder` will ARRAYREF and HASHREF tags when the item contains less than 16 items, and and is not referenced more than once. This flag will override this optimization and use a standard REFN ARRAY style tag output. This is primarily useful for producing canonical output and for testing Sereal itself.

See "CANONICAL REPRESENTATION" for why you might want to use this, and for the various caveats involved.

sort_keys

Normally `Sereal::Encoder` will output hashes in whatever order is convenient, generally that used by perl to actually store the hash, or whatever order was returned by a tied hash.

If this option is enabled then the Encoder will sort the keys before outputting them. It uses more memory, and is quite a bit slower than the default.

Generally speaking this should mean that a hash and a copy should produce the same output. Nevertheless the user is warned that Perl has a way of "morphing" variables on use, and some of its rules are a little arcane (for instance utf8 keys), and so two hashes that might appear to be the same might still produce different output as far as Sereal is concerned.

As of 3.006_007 (prerelease candidate for 3.007) the sort order has been changed to the following: order by length of keys (in bytes) ascending, then by byte order of the raw underlying string, then by utf8ness, with non−utf8 first. This order was chosen because it is the most efficient to implement, both in terms of memory and time. This sort order is enabled when sort_keys is set to 1.

You may also produce output in Perl "cmp" order, by setting sort_keys to 2. And for backwards compatibility you may also produce output in reverse Perl "cmp" order by setting sort_keys to 3. Prior to

3.006_007 this was the only sort order possible, although it was not explicitly defined what it was.

Note that comparatively speaking both of the ‘‘cmp’’ sort orders are slow and memory inefficient. Unless you have a really good reason stick to the default which is fast and as lean as possible.

Unless you are concerned with ‘‘cross process canonical representation’’ then it doesn’t matter what option you choose.

See ‘‘CANONICAL REPRESENTATION’’ for why you might want to use this, and for the various caveats involved.

no_shared_hashkeys

When the `no_shared_hashkeys` option is set to a true value, then the encoder will disable the detection and elimination of repeated hash keys. This only has an effect for serializing structures containing hashes. By skipping the detection of repeated hash keys, performance goes up a bit, but the size of the output can potentially be much larger.

Do not disable this unless you have a reason to.

dedupe_strings

If this is option is enabled/true then Sereal will use a hash to encode duplicates of strings during serialization efficiently using (internal) backreferences. This has a performance and memory penalty during encoding so it defaults to off. On the other hand, data structures with many duplicated strings will see a significant reduction in the size of the encoded form. Currently only strings longer than 3 characters will be deduped, however this may change in the future.

Note that Sereal will perform certain types of deduping automatically even without this option. In particular class names and hash keys (see also the `no_shared_hashkeys` setting) are deduped regardless of this option. Only enable this if you have good reason to believe that there are many duplicated strings as values in your data structure.

Use of this option does not require an upgraded decoder (this option was added in Sereal::Encoder 0.32). The deduping is performed in such a way that older decoders should handle it just fine. In other words, the output of a Sereal **decoder** should not depend on whether this option was used during **encoding**. See also below: *aliased_dedupe_strings*.

aliased_dedupe_strings

This is an advanced option that should be used only after fully understanding its ramifications.

This option enables a mode of operation that is similar to *dedupe_strings* and if both options are set, *aliased_dedupe_strings* takes precedence.

The behaviour of *aliased_dedupe_strings* differs from *dedupe_strings* in that the duplicate occurrences of strings are emitted as Perl language level **aliases** instead of as Sereal-internal backreferences. This means that using this option actually produces a different output data structure when decoding. The upshot is that with this option, the application using (decoding) the data may save a lot of memory in some situations but at the cost of potential action at a distance due to the aliasing.

Beware: The test suite currently does not cover this option as well as it probably should. Patches welcome.

protocol_version

Specifies the version of the Sereal protocol to emit. Valid are integers between 1 and the current version. If not specified, the most recent protocol version will be used. See also `use_protocol_v1`:

It is strongly advised to use the latest protocol version outside of migration periods.

use_protocol_v1

This option is deprecated in favour of the `protocol_version` option (see above).

If set, the encoder will emit Sereal documents following protocol version 1. This is strongly discouraged except for temporary compatibility/migration purposes.

INSTANCE METHODS

encode

Given a Perl data structure, serializes that data structure and returns a binary string that can be turned back into the original data structure by Sereal::Decoder. The method expects a data structure to serialize as first argument, optionally followed by a header data structure.

A header is intended for embedding small amounts of meta data, such as routing information, in a document that allows users to avoid deserializing main body needlessly.

encode_to_file

```
Sereal::Encoder->encode_to_file($file,$data,$append);
$encoder->encode_to_file($file,$data,$append);
```

Encode the data specified and write it the named file. If `$append` is true then the written data is appended to any existing data, otherwise any existing data will be overwritten. Dies if any errors occur during writing the encoded data.

EXPORTABLE FUNCTIONS

sereal_encode_with_object

The functional interface that is equivalent to using `encode`. Takes an encoder object reference as first argument, followed by a data structure and optional header to serialize.

This functional interface is marginally faster than the OO interface since it avoids method resolution overhead and, on sufficiently modern Perl versions, can usually avoid subroutine call overhead.

encode_sereal

The functional interface that is equivalent to using `new` and `encode`. Expects a data structure to serialize as first argument, optionally followed by a hash reference of options (see documentation for `new()`).

This function cannot be used for encoding a data structure with a header. See `encode_sereal_with_header_data`.

This functional interface is significantly slower than the OO interface since it cannot reuse the encoder object.

encode_sereal_with_header_data

The functional interface that is equivalent to using `new` and `encode`. Expects a data structure and a header to serialize as first and second arguments, optionally followed by a hash reference of options (see documentation for `new()`).

This functional interface is significantly slower than the OO interface since it cannot reuse the encoder object.

PERFORMANCE

See Sereal::Performance for detailed considerations on performance tuning. Let it just be said that:

If you care about performance at all, then use "sereal_encode_with_object" or the OO interface instead of "encode_sereal". It's a significant difference in performance if you are serializing small data structures.

The exact performance in time and space depends heavily on the data structure to be serialized. Often there is a trade-off between space and time. If in doubt, do your own testing and most importantly ALWAYS TEST WITH REAL DATA. If you care purely about speed at the expense of output size, you can use the `no_shared_hashkeys` option for a small speed-up. If you need smaller output at the cost of higher CPU load and more memory used during encoding/decoding, try the `dedupe_strings` option and enable Snappy compression.

For ready-made comparison scripts, see the *author_tools/bench.pl* and *author_tools/dbench.pl* programs that are part of this distribution. Suffice to say that this library is easily competitive in both time and space efficiency with the best alternatives.

FREEZE/THAW CALLBACK MECHANISM

This mechanism is enabled using the `freeze_callbacks` option of the encoder. It is inspired by the equivalent mechanism in CBOR::XS and differs only in one minor detail, explained below. The general mechanism is documented in the *A GENERIC OBJECT SERIALIATION PROTOCOL* section of Types::Serializer. Similar to CBOR using `CBOR`, Sereal uses the string `Sereal` as a serializer identifier for the callbacks.

The one difference to the mechanism as supported by CBOR is that in Sereal, the `FREEZE` callback must return a single value. That value can be any data structure supported by Sereal (hopefully without causing infinite recursion by including the original object). But `FREEZE` can't return a list as with CBOR. This should not be any practical limitation whatsoever. Just return an array reference instead of a list.

Here is a contrived example of a class implementing the `FREEZE` / `THAW` mechanism.

```
package
  File;

use Moo;

has 'path' => (is => 'ro');
has 'fh' => (is => 'rw');

# open file handle if necessary and return it
sub get_fh {
  my $self = shift;
  # This could also be done with fancier Moo(se) syntax
  my $fh = $self->fh;
  if (not $fh) {
    open $fh, "<", $self->path or die $!;
    $self->fh($fh);
  }
  return $fh;
}

sub FREEZE {
  my ($self, $serializer) = @_;
  # Could switch on $serializer here: JSON, CBOR, Sereal, ...
  # But this case is so simple that it will work with ALL of them.
  # Do not try to serialize our file handle! Path will be enough
  # to recreate.
  return $self->path;
}

sub THAW {
  my ($class, $serializer, $data) = @_;
  # Turn back into object.
  return $class->new(path => $data);
}
```

Why is the `FREEZE`/`THAW` mechanism important here? Our contrived `File` class may contain a file handle which can't be serialized. So `FREEZE` not only returns just the path (which is more compact than encoding the actual object contents), but it strips the file handle which can be lazily reopened on the other side of the serialization/deserialization pipe. But this example also shows that a naive implementation can easily end up with subtle bugs. A file handle itself has state (position in file, etc). Thus the deserialization in the above example won't accurately reproduce the original state. It can't, of course, if it's deserialized in a different environment anyway.

THREAD-SAFETY

`Sereal::Encoder` is thread-safe on Perl's 5.8.7 and higher. This means ''thread-safe'' in the sense that if you create a new thread, all `Sereal::Encoder` objects will become a reference to undef in the new thread. This might change in a future release to become a full clone of the encoder object.

CANONICAL REPRESENTATION

You might want to compare two data structures by comparing their serialized byte strings. For that to work reliably the serialization must take extra steps to ensure that identical data structures are encoded into identical serialized byte strings (a so-called ''canonical representation'').

Unfortunately in Perl there is no such thing as a ''canonical representation''. Most people are interested in ''structural equivalence'' but even that is less well defined than most people think. For instance in the following example:

```
my $array1= [ 0, 0 ];
my $array2= do {
    my $zero= 0;
    sub{ \@_ }->($zero,$zero);
};
```

the question of whether `$array1` is structurally equivalent to `$array2` is a subjective one. Sereal for instance would **NOT** consider them equivalent but `Test::Deep` would. There are many examples of this in Perl. Simply stringifying a number technically changes the scalar. Storable would notice this, but Sereal generally would not.

Despite this as of 3.002 the Sereal encoder supports a ''canonical'' option which will make a ''best effort'' attempt at producing a canonical representation of a data structure. This mode is actually a combination of several other modes which may also be enabled independently, and as and when we add new options to the encoder that would assist in this regard then the `canonical` will also enable them. These options may come with a performance penalty so care should be taken to read the Changes file and test the performance implications when upgrading a system that uses this option.

It is important to note that using canonical representation to determine if two data structures are different is subject to false-positives. If two Sereal encodings are identical you can generally assume that the two data structures are functionally equivalent from the point of view of normal Perl code (XS code might disagree). However if two Sereal encodings differ the data structures may actually be functionally equivalent. In practice it seems the the false-positive rate is low, but your milage may vary.

Some of the issues with producing a true canonical representation are outlined below:

Sereal doesn't order the hash keys by default.
: This can be enabled via the `sort_keys`, which is itself enabled by `canonical` option.

Sereal output is sensitive to refcounts
: This can be somewhat mitigated by the use of `canonical_refs`, see above.

There are multiple valid Sereal documents that you can produce for the same Perl data structure.
: Just sorting hash keys is not enough. Some of the reasons are outlined below. These issues are especially relevant when considering language interoperability.

  PAD bytes
  : A trivial example is PAD bytes which mean nothing and are skipped. They mostly exist for encoder optimizations to prevent certain nasty backtracking situations from becoming O(n) at the cost of one byte of output. An explicit canonical mode would have to outlaw them (or add more of them) and thus require a much more complicated implementation of refcount/weakref handing in the encoder while at the same time causing some operations to go from O(1) to a full memcpy of everything after the point of where we backtracked to. Nasty.

  COPY tag
  : Another example is COPY. The COPY tag indicates that the next element is an identical copy of a previous element (which is itself forbidden from including COPY's other than for class names). COPY is purely internal. The Perl/XS implementation uses it to share hash keys and class names.

One could use it for other strings (theoretically), but doesn't for time-efficiency reasons. We'd have to outlaw the use of this (significant) optimization of canonicalization.

REF representation
: Sereal represents a reference to an array as a sequence of tags which, in its simplest form, reads *REF, ARRAY `$array_length` TAG1 TAG2 ....* The separation of "REF" and "ARRAY" is necessary to properly implement all of Perl's referencing and aliasing semantics correctly. Quite frequently, however, your array is only referenced once and plainly so. If it's also at most 15 elements long, Sereal optimizes all of the "REF" and "ARRAY" tags, as well as the length into a special one byte ARRAYREF tag. This is a very significant optimization for common cases. This, however, does mean that most arrays up to 15 elements could be represented in two different, yet perfectly valid forms. ARRAYREF would have to be outlawed for a properly canonical form. The exact same logic applies to HASH vs. HASHREF. This behavior can be overridden by the `canonical_refs` option, which disables use of HASHREF and ARRAYREF.

Numeric representation
: Similar to how Sereal can represent arrays and hashes in a full and a compact form. For small integers (between −16 and +15 inclusive), Sereal emits only one byte including the encoding of the type of data. For larger integers, it can use either variants (positive only) or zigzag encoding, which can also represent negative numbers. For a canonical mode, the space optimizations would have to be turned off and it would have to be explicitly specified whether variant or zigzag encoding is to be used for encoding positive integers.

  Perl may choose to retain multiple representations of a scalar. Specifically, it can convert integers, floating point numbers, and strings on the fly and will aggressively cache the results. Normally, it remembers which of the representations can be considered canonical, that means, which can be used to recreate the others reliably. For example, `0` and `"0"` can both be considered canonical since they naturally transform into each other. Beyond intrinsic ambiguity, there are ways to trick Perl into allowing a single scalar to have distinct string, integer, and floating point representations that are all flagged as canonical, but can't be transformed into each other. These are the so-called dualvars. Sereal cannot represent dualvars (and that's a good thing).

  Floating point values can appear to be the same but serialize to different byte strings due to insignificant 'noise' in the floating point representation. Sereal supports different floating point precisions and will generally choose the most compact that can represent your floating point number correctly.

There's also a few cases where Sereal will produce different documents for values that you might think are the same thing, because if you e.g. compared them with `eq` or == in perl itself would think they were equivalent. However for the purposes of serialization they're not the same value.

A good example of these cases is where Test::Deep and Sereal's canonical mode differ. We have tests for some of these cases in *t/030_canonical_vs_test_deep.t*. Here's the issues we've noticed so far:

Sereal considers ASCII strings with the UTF−8 flag to be different from the same string without the UTF−8 flag
: Consider:

```
my $language_code = "en";
```

  v.s.:

```
my $language_code = "en";
utf8::upgrade($en);
```

  Sereal's canonical mode will encode these strings differently, as it should, since the UTF−8 flag will be passed along on interpolation.

  But this can be confusing if you're just getting some user-supplied ASCII strings that you may inadvertently toggle the UTF−8 flag on, e.g. because you're comparing an ASCII value in a database to a value submitted in a UTF−8 web form.

Sereal will encode strings that look like numbers as strings, unless they've been used in numeric context

I.e. these values will be encoded differently, respectively:

```
my $IV_x = "12345";
my $IV_y = "12345" + 0;
my $NV_x = "12.345";
my $NV_y = "12.345" + 0;
```

But as noted above something like Test::Deep will consider these to be the same thing.

We might produce certain aggressive flags to the canonical mode in the future to deal with this. For the cases noted above some combination of turning the UTF–8 flag on on all strings, or stripping it from strings that have it but are ASCII-only would ''work'', similarly we could scan strings to see if they match `looks_like_number()` and if so numify them.

This would produce output that either would be a lot bigger (having to encode all numbers as strings), or would be more expensive to generate (having to scan strings for numeric or non-ASCII context), and for some cases like the UTF–8 flag munging wouldn't be suitable for general use outside of canonicialization.

Often, people don't actually care about ''canonical'' in the strict sense required for real *identity* checking. They just require a best-effort sort of thing for caching. But it's a slippery slope!

In a nutshell, the `canonical` option may be sufficient for an application which is simply serializing a cache key, and thus there's little harm in an occasional false-negative, but think carefully before applying Sereal in other use-cases.

KNOWN ISSUES

Strings Or Numbers

Perl does not make a strong distinction between strings and numbers, and from an internal point of view it can be difficult to tell what the ''right'' representation is for a given variable.

Sereal tries to not be lossy. So if it detects that the string value of a var, and the numeric value are different it will generally round trip the *string* value. This means that ''special'' strings often used in Perl function returns, like ''0 but true'', and ''0e0'', will round trip in a way that their normal Perl semantics are preserved. However this also means that ''non canonical'' values, like '' 100 '', which will numify as 100 without warnings, will round trip as their string values.

Perl also has some operators, the binary operators, ^, | and &, which do different things depending on whether their arguments had been used in numeric context as the following examples show:

```
perl -le'my $x="1"; $i=int($x); print unpack "H*", $x ^ "1"'
30

perl -le'my $x="1"; print unpack "H*", $x ^ "1"'
00

perl -le'my $x=" 1 "; $i=int($x); print unpack "H*", $x ^ "1"'
30

perl -le'my $x=" 1 "; print unpack "H*", $x ^ "1"'
113120
```

Sereal currently cannot round trip this property properly.

An extreme case of this problem is that of ''dualvars'', which can be created using the **Scalar::Util::dualvar()** function. This function allows one to create variables which have string and integer values which are completely unrelated to each other. Sereal currently will choose the *string* value when it detects these items.

It is possible that a future release of the protocol will fix these issues.

BUGS, CONTACT AND SUPPORT

For reporting bugs, please use the github bug tracker at <http://github.com/Sereal/Sereal/issues>.

For support and discussion of Sereal, there are two Google Groups:

Announcements around Sereal (extremely low volume): <https://groups.google.com/forum/?fromgroups#!forum/sereal−announce>

Sereal development list: <https://groups.google.com/forum/?fromgroups#!forum/sereal−dev>

AUTHORS AND CONTRIBUTORS

Yves Orton <demerphq@gmail.com>

Damian Gryski

Steffen Mueller <smueller@cpan.org>

Rafaël Garcia-Suarez

Ævar Arnfjörð Bjarmason <avar@cpan.org>

Tim Bunce

Daniel Dragan <bulkdd@cpan.org> (Windows support and bugfixes)

Zefram

Borislav Nikolov

Ivan Kruglov <ivan.kruglov@yahoo.com>

Some inspiration and code was taken from Marc Lehmann's excellent JSON::XS module due to obvious overlap in problem domain. Thank you!

ACKNOWLEDGMENT

This module was originally developed for Booking.com. With approval from Booking.com, this module was generalized and published on CPAN, for which the authors would like to express their gratitude.

COPYRIGHT AND LICENSE