
libENI Documentation

Release stable

Sep 09, 2019

Contents

1	Contents	3
1.1	Getting Started	3
1.2	Developer Guide	5
1.3	Documentation	9
1.4	Contributing to libENI	10

Official C++ implementation of [libENI](#), which is part of the [Lity](#) project.

1.1 Getting Started

1.1.1 Download the Prebuilt libENI

See [libENI releases](#) for the latest release.

- *libeni-dev*: for ENI operation developers
- *libeni*: for general ENI users.

Releases

Version	libeni-dev	libeni
v1.3.6	Ubuntu 16.04, CentOS 7	Ubuntu 16.04, CentOS 7
v1.2.x	Ubuntu 16.04, CentOS 7	Ubuntu 16.04, CentOS 7
v1.2.0	Ubuntu 16.04, CentOS 7	Ubuntu 16.04, CentOS 7

Prerequisites

libeni-dev	libeni
<ul style="list-style-type: none">• Boost >= 1.58• OpenSSL >= 1.0.2	<ul style="list-style-type: none">• OpenSSL >= 1.0.2

See Prerequisites for platform specific prerequisites guide.

Install

```
tar zxvf libeni.tgz --strip-components 1 -C ${LIBENI_PATH}
```

Validate the Shared Libraries

```
cd ${LIBENI_PATH}/lib
sha512sum -c *.sha512
```

You should get a list of OKs if all libraries are good.

```
eni_crypto.so: OK
eni_reverse.so: OK
eni_scrypt.so: OK
```

Test Manually

See Testing Prebuilt ENI Operations for how to test the prebuilt shared libraries of ENI operations.

1.1.2 Build From Source

Prerequisites

- Boost \geq 1.58
- CMake \geq 3.1
- OpenSSL \geq 1.0.2
- SkyPat \geq 3.1.1 (see [SkyPat releases](#))

Download Source Code

```
git clone https://github.com/CyberMiles/libeni.git ${LIBENI_PATH}
```

Build with CMake

```
cd ${LIBENI_PATH}
mkdir build
cd build
cmake ..
make
```

Run Tests

In your build directory, run `ctest`. The result looks like the below.


```

Test project ${LIBENI_PATH}/build
  Start 1: eni_reverse_checksum_test
1/13 Test #1: eni_reverse_checksum_test ..... Passed    0.00 sec
  Start 2: eni_crypto_checksum_test
2/13 Test #2: eni_crypto_checksum_test ..... Passed    0.00 sec
  Start 3: crypto_unittests
3/13 Test #3: crypto_unittests ..... Passed    0.01 sec
  Start 4: eni_scrypt_checksum_test
4/13 Test #4: eni_scrypt_checksum_test ..... Passed    0.00 sec
  Start 5: scrypt_unittests
5/13 Test #5: scrypt_unittests ..... Passed    0.01 sec
  Start 6: t0000-smoke
6/13 Test #6: t0000-smoke ..... Passed    0.00 sec
  Start 7: t0005-tools-eni-scrypt
7/13 Test #7: t0005-tools-eni-scrypt ..... Passed    0.01 sec
  Start 8: t0004-tools-eni-crypto
8/13 Test #8: t0004-tools-eni-crypto ..... Passed    0.02 sec
  Start 9: t0001-testlib
9/13 Test #9: t0001-testlib ..... Passed    0.01 sec
  Start 10: t0002-examples-eni-reverse
10/13 Test #10: t0002-examples-eni-reverse ..... Passed    0.01 sec
  Start 11: consensus_tests
11/13 Test #11: consensus_tests ..... Passed    0.07 sec
  Start 12: malformed_consensus_tests
12/13 Test #12: malformed_consensus_tests ..... Passed    0.03 sec
  Start 13: unittests
13/13 Test #13: unittests ..... Passed    1.37 sec

100% tests passed, 0 tests failed out of 13

Label Time Summary:
auto          = 0.06 sec (8 tests)
checksum      = 0.00 sec (3 tests)
regression    = 0.13 sec (6 tests)
unittest      = 1.40 sec (3 tests)

Total Test time (real) = 1.57 sec

```

See *Testing/Temporary/LastTest.log* for the detailed output of all tests.

1.2 Developer Guide

In this tutorial, we will guide you through how to create new ENI operations with libENI in C++.

1.2.1 Prerequisites

In order to build your ENI operations, you need to install *libeni-dev* first.

See [Getting Started](#) for more information.

1.2.2 Contents

Implement an ENI Operation

Here, we use *examples/eni/reverse* as an example. In this example, we will create an ENI operation called *reverse* that takes a string, and returns the reversed string.

The below code piece shows how developers use this ENI operation when writing a contract in Solidity.

```
string memory reversed;  
reversed = eni("reverse", "The string to be reversed.");
```

Subclass `EniBase`

In order to implement an ENI operation, you need to `#include <eni.h>`, create a subclass of `eni::EniBase`, and implement the following functions.

0. A constructor that takes a string as its parameter. Remember to pass the string to the constructor of the superclass, `eni::EniBase`, which will convert the raw string into a `json::Array` containing the arguments for your ENI operation.
1. A destructor.
2. Three pure virtual functions, which should be implement privately.
 - `parse` to parse the arguments.
 - `gas` to calculate gas consumption from the arguments.
 - `run` to execute your ENI operation with the arguments.

```
#include <eni.h>  
class Reverse : public eni::EniBase {  
public:  
    Reverse(const std::string& pArgStr)  
        : eni::EniBase(pArgStr) { ... }  
  
    ~Reverse() { ... }  
  
private:  
    bool parse(const json::Array& pArgs) override { ... }  
  
    eni::Gas gas() const override { ... }  
  
    bool run(json::Array& pRetVal) override { ... }  
};
```

Parse Arguments

The `parse` function takes a `json::Array` containing the arguments given to your ENI operation. To ensure the other two functions `gas` and `run` process the arguments in the same way, please validate, preprocess, and store the arguments into member variables in the `parse` function.

The `parse` function should return `true` when all arguments are good, and return `false` otherwise. (i.e. when the given arguments are not correct, e.g., lacking arguments, or wrong type).

In this example, the `json::Array` constructed by `eni::EniBase` contains only the argument string for ENI operation *reverse*.

```
["The string to be reversed."]
```

Here we just take the first argument and convert it to a string.

```
class Reverse : public eni::EniBase {
    ...
private:
    bool parse(const json::Array& pArgs) override {
        m_Str = pArgs[0].toString();
        return true;
    }

    std::string m_Str;
};
```

Check the documentation to see more detail about how arguments are converted into a `json::Array`.

Estimate Gas Consumption

Before your ENI operation is run, you need to estimate how much gas it will cost. Override the pure virtual function `gas`, and return your estimated gas cost.

In this example, we use the string length as its gas consumption.

```
class Reverse : public eni::EniBase {
    ...
private:
    eni::Gas gas() const override {
        return m_Str.length();
    }
};
```

Return 0 when error occurs (e.g., gas is incalculable).

Execute the Operation

Override the pure virtual function `run`, and push the result of your ENI operation back into the `json::Array`.

```
class Reverse : public eni::EniBase {
    ...
private:
    bool run(json::Array& pRetVal) override {
        std::string ret(m_Str.rbegin(), m_Str.rend());
        pRetVal.emplace_back(ret);
        return true;
    }
};
```

Return `true` only when your ENI operation is successfully executed.

Export the ENI Operation with C Interface

Your ENI operation will be called via its C interface, so be sure to export the C interface with `ENI_C_INTERFACE(OP, CLASS)`, where *OP* is your ENI operation name (i.e., *reverse* in this example), and

CLASS is the name of implemented class (i.e., *Reverse* in this example).

```
ENI_C_INTERFACE(reverse, Reverse)
```

Related Guides

Next: *Build ENI Operations Into a Shared Library*.

Build ENI Operations Into a Shared Library

Please add these flags `-std=c++11 -fPIC` when compiling your ENI operation into a shared library. See [GCC Option Summary](#) for explanation to these flags.

Specify the path to libENI headers with `-I${LIBENI_PATH}/include`.

You might also want to link to libENI by specifying the path `-L${LIBENI_PATH}/lib`, and the library name `-leni`.

Here is an example Makefile for *examples/eni/reverse*. Please be aware that the flags and commands might differ if you're using different compilers.

```
CPPFLAGS=-I${LIBENI_PATH}/include
CXXFLAGS=-std=c++11 -fPIC
LDFLAGS=-L${LIBENI_PATH}/lib
LDADD=-leni

all:
    g++ ${CPPFLAGS} ${CXXFLAGS} ${LDFLAGS} -shared -oeni_reverse.so eni_reverse.cpp
    ↪ ${LDADD}
```

Related Guides

Next: *Test Your ENI Operations*. Previous: *Implement an ENI Operation*.

Test Your ENI Operations

Test From `EniBase` Interface

Your ENI operations will only be accessed from the two public member functions of `eni::EniBase`.

- `Gas getGas()` should return the gas cost of your ENI operation.
- `char* start()` should run your ENI operation and return the results in JSON format.

You may test your subclass through these two public functions.

```
eni::EniBase* functor = new Reverse("[\"Hello World\"]");
ASSERT_NE(functor, nullptr);
EXPECT_EQ(functor->getGas(), 12);
char* ret = functor->start();
EXPECT_EQ(::strcmp(ret, "[\"!dlroW olleH\"]"), 0);
free(ret);
delete functor;
```

Test From Shared Library Interface

Setup Environment

Make sure libENI can be found in your environment. See [Getting Started](#) for how to install libENI.

You might want to try the following settings if libENI is installed but not found in your environment.

```
PATH=${PATH}:${LIBENI_PATH}/bin
LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:${LIBENI_PATH}/lib
```

Tools for Testing

Related Guides

Previous: *Build ENI Operations Into a Shared Library.*

1.3 Documentation

1.3.1 Types

ENI Types

These types are provided to be coherent with [primitive types of Lity \(Solidity\)](#).

ENI Integers

All integer types in ENI is implemented using [boost::multiprecision::number](#). Some of them are aliases for types predefined in [boost::multiprecision::cpp_int](#).

Integer Type	Size (bits)	Signed	Note
<code>eni::s256</code>	256	✓	Alias for <code>boost::multiprecision::int256_t</code> .
<code>eni::Int</code>	256	✓	Alias for <code>eni::s256</code> .
<code>eni::u256</code>	256		Alias for <code>boost::multiprecision::uint256_t</code> .
<code>eni::u160</code>	160		Size of an Ethereum address.
<code>eni::u128</code>	128		Alias for <code>boost::multiprecision::uint128_t</code> .
<code>eni::u64</code>	64		
<code>eni::UInt</code>	256		Alias for <code>eni::u256</code> .

Operations on ENI Integers

See [the documentation for boost::multiprecision::number](#) for supported operations.

Suggested Use of ENI Integers

This section does not exist yet. (‘-l-‘)

ENI Boolean

`eni::Bool` is an alias for C++ `bool`.

ENI Address

`eni::Address` is an alias for `eni::u160` (20 bytes, size of an Ethereum address).

Convert ENI Types to C++ String

`eni::to_string` uses `boost::lexical_cast` internally to convert ENI types to `std::string`.

All ENI integers, `eni::Bool`, `eni::Address` are supported.

```
std::string to_string(enl::TypeName);
```

Usage

```
eni::Int int32max(2147483647);
std::string s = eni::to_string(int32max); // "2147483647"

eni::Bool bTrue(true);
std::string t = eni::to_string(bTrue);    // "true"
```

Abstract Data Types

See `eni::Vector` and `eni::StringMap`.

This section does not exist yet. (‘-l_’)

JSON Types

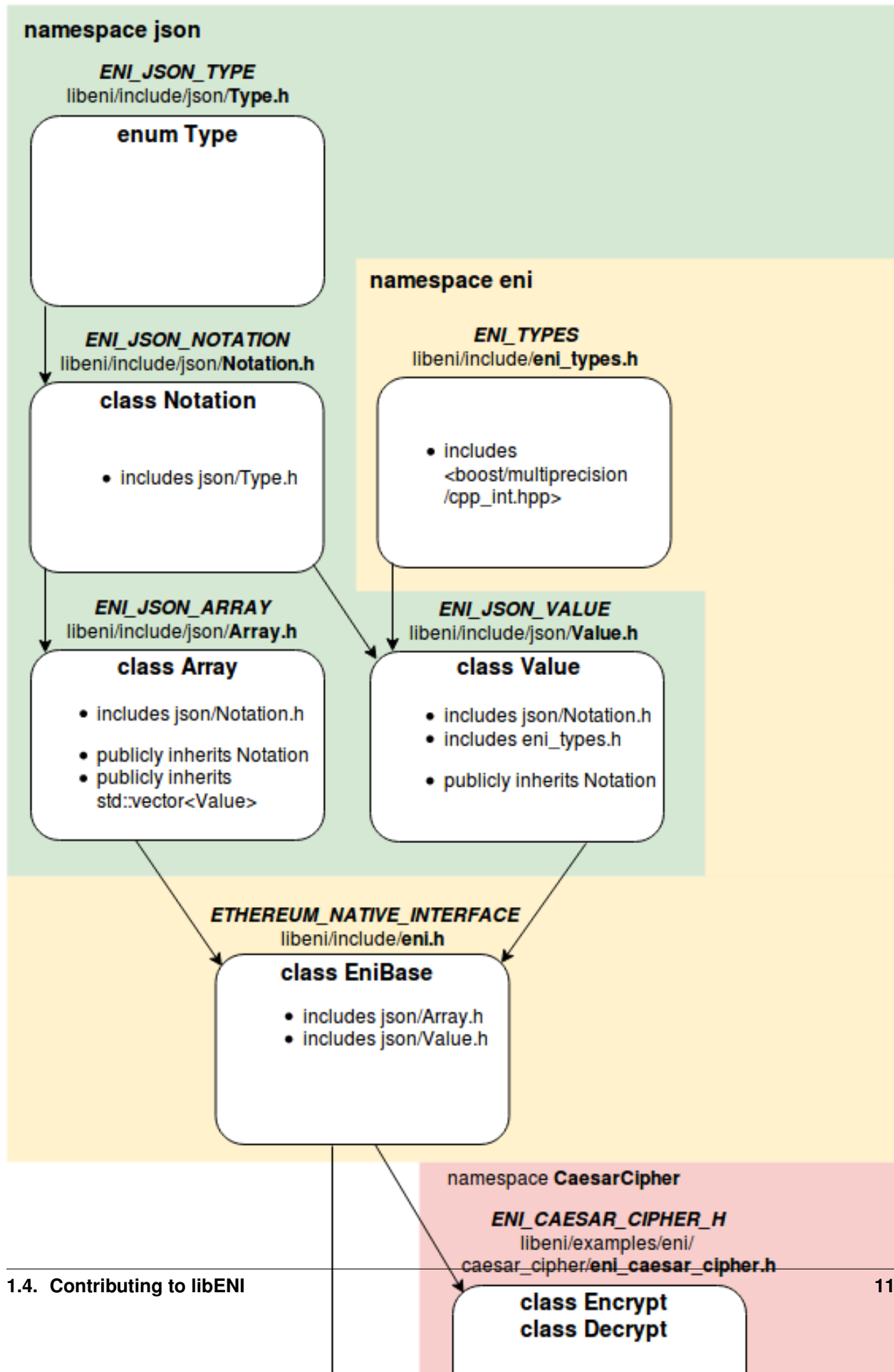
This section does not exist yet. (‘-l_’)

1.4 Contributing to libENI

See *Getting Started* for how to build libENI.

1.4.1 Overview of libENI Code

The following image illustrates the libENI code hierarchy and the use of namespaces.



Directory Structure

Path	Description
docs/	Documentations.
examples/	Examples of how to use libENI.
include/	Header files for libENI.
lib/	Implementations for libENI.
test/	All tests for libENI and its examples.
tools/	Tools and modules for libENI.

1.4.2 Report an Issue

Please provide the following information as much as possible.

- The version (commit-ish) your using.
- Your platform, environment setup, etc.
- Steps to reproduce the issue.
- Your expected result of the issue.
- Current result of the issue.

1.4.3 Create a Pull Request

- Fork from the *master* branch.
- Avoid to create merge commits when you update from *master* by using `git rebase` or `git pull --rebase` (instead of `git merge`).
- Add test cases for your pull request if you're proposing new features or major bug fixes.
- Build and test locally before submit your pull request. See [Getting Started](#) for how to test libENI.

Please try to follow the existing coding style of libENI (although it is neither well-styled nor documented at this moment), which is basically based on [LLVM coding standards](#).