ccc Documentation

Release latest

Sep 18, 2019

Documentation

CCC is an **EVM** compiler which tries to offer the minimal required features to write *smart contracts* in an old fashion **C** style.

CCC is the name of the language as well.

CHAPTER 1

Introduction

After exploring the available **EVM** compilers offered by the *open source* community, being not able to find one of them capable to satisfy me, I decided to undertake this challenging project.

1.1 Philosophy

Thinking just a while to the **EVM** we all can agree that it is a really *small environment*, that's why (in my opinion) an **EVM** *dedicated compiler* should offer the less as possible features to comfortably write *smart contracts* while keeping the generated *opcode* the more thin as possible.

1.2 Warning

I learned everithing I know about **EVM** reading from the internet or through reverse engineering. I have no way to say if what I read is wrong or outdated, I can't say if there are other ways to do what I discovered: *do not trust me!* If you find something wrong, outdated, or false for any other reasons, please do not hesitate to report it on github.

1.3 Assumptions

In this documentation there are several *assumptions* as the following example.

Assumption

1 + 1 = 3

Assumptions can be read in two ways: decontextualized or in the context of **CCC**. If reading one of them you find it wrong, please consider the previous *Warning*. Regardless of that, in the context of **CCC** they can be taken as *the truth* due to the fact that **CCC** is written respecting them as *the truth* should be respected.

CHAPTER 2

Installation

CCC is distributed through npm.

To install the *Command Line Interface* issue following command:

```
$ sudo npm install -g cccompiler
```

To install only the *Node.js module* to use it in your own build system, issue following command:

```
$ npm install -s cccompiler
```

2.1 Source code

Source repository is on github.

CHAPTER 3

Compilation

The compilation process takes in input only one file (many other files can be included through the `#include` *preprocessor directive*) and gives in output a *JSON Object* (called *Output JSON Object*) containing all requested information that can be provided. The entire process is composed by a set of *phases*, executed in a sequence and everyone of which produces an output for each **contract** compiled. Starting *phase* can be eventually specified by *Compilation options*. If during the execution of a *phase* an **error** is generated, the process stops at the end of that *phase* and no output for that *phase* is provided (obviously no output for subsequent *phases* is provided neither).

3.1 Compilation phases

Here is the list of all *phases*, in the order they are executed. The input of the first executed *phase* is the content of the file provided in input to the whole process, while for all subsequent *phases* the output of previously executed *phase* is used as input.

3.1.1 Preprocess

Runs the **CCC preprocessor**. Both *input format* and *output format* are **CCC**.

3.1.2 Compile

Runs the **CCC compiler**. This *phase* produces two outputs: the abi and the **assembly** representation of the compiled contract(s) which is eventually used as input for next *phase*. The expected *input format* is **CCC**.

3.1.3 Assemble

Runs the **CCC assembler** which provides the **opcode** representation of the contract(s). Both *input format* and *output format* are **assembly**.

3.1.4 Opcode

Translates the **assembly** in the **opcodes**: Basically it resolves the *assembly labels* in their realtive or absolute address.

3.1.5 Translate

Literally translates the **opcodes** in the hexadecimal representation of the **bytecode** of the contract(s), ready to be deployed on *blockchain*.

3.2 Output JSON Object

This is a *JSON Object* with four keys. Follows an example.

```
{
    "contracts": {
        "FirstContract": {
            "abi": [...],
            "assembly": [...],
            "bin": [...],
            "preprocessed": [...],
            "opcodes": [...]
        },
        ...
    },
    "errors": [],
    "messages": [],
    "warnings": []
}
```

3.2.1 contracts

This is a *JSON Object* where the *keys* are the names of all compiled contracts and the *values* are the output of all *phase* run during the compilation process.

3.2.2 errors

The array of all encoutered *errors*, eventually the empty array `[]`.

3.2.3 messages

The array of all generated *messages*, eventually the empty array `[]`. This is merely the merge of all *errors* and *warnings*.

3.2.4 warnings

The array of all encoutered *warnings*, eventually the empty array `[]`.

3.3 Compilation options

Some options may conflict due to the fact some one of them specify the *input format* of the file or they define conflicting starting and ending *phase*.

3.3.1 assemble

`Boolean` - If `true` specifies that *input format* is in **assembly** and starting *phase* is **Assemble**.

3.3.2 assembly

`Boolean` - If `true` includes the generated **assembly** in the output, specifies that *input format* is in **CCC** and starting *phase* is **Compile**.

3.3.3 define

`Object` - Specifies a set of predefined `#define` *macros*. Each *key* is the name of the *macro* and relative *value* is the value of the *macro*. The *values* **must** be of type `String`, eventually the empty string `""`.

3.3.4 opcode

`Boolean` - If `true` includes the generated **opcode** in the output.

3.3.5 preprocess

`Boolean` - If `true` includes the **preprocessor** result in the output, specifies that *input format* is in **CCC**.

CHAPTER 4

Command Line Interface

CCC *cli* is simply a *shell interface* to the *Node.js module*. Please refer to *Compilation* to understand how the process works.

4.1 Usage

```
ccc [options] file
```

4.2 Options

For more details about about options please refere to *Compilation options*.

4.2.1 -A –assembly

Includes the generated **assembly** in the output; assumes *input format* is in **CCC**.

4.2.2 -a –assemble

Assemble; assumes *input format* is in **assembly**.

4.2.3 -D<macro>[=<value>]

Defines `<macro>` eventually with its `<value>`. Can be used multiple times to define more *macros*.

4.2.4 -h –help

Prints a quick reference help screen and exits.

4.2.5 -O –opcode

Includes the generated **opcode** in the output.

4.2.6 -o <filename>

The output `<filename>`. If omitted defaults *stdout*.

4.2.7 -p –preprocess

Includes the **preprocessor** result in the output; assumes *input format* is in **CCC**.

4.2.8 -v –version

Prints **ccc** version and exits.

CHAPTER 5

Node.js module

5.1 Usage

```
var ccc = require('cccompiler');
var res = ccc(filename, options);

console.log(res.contracts.contractName.preprocessed);
```

Where `filename` is the name of the file to compile, `options` are described in *Compilation options* and return value is *Output JSON Object*.

CHAPTER 6

Smart Contract

6.1 ABI Interface

For **ABI** specification please refer to Solidity ABI specification.