Adding a Precompile

⚠ OP Stack Hacks are explicitly things that you can do with the OP Stack that arenot currently intended for production use.

OP Stack Hacks are not for the faint of heart. You will not be able to receive significant developer support for OP Stack Hacks — be prepared to get your hands dirty and to work without support. One possible use of OP Stack is to run an EVM with a new precompile for operations to speed up calculations that are not currently supported. In this tutorial, you'll make a simple precompile that returns a constant value if it's called with four or less bytes, or an error if it is called with more than that.

To create a new precompile, the file to modify isop-geth/core/vm/contracts.go (opens in a new tab).

Add toPrecompiledContractsBerlin

on line 82 (or a later fork, if the list of precompiles changes again)

- add a structure named after your new precompile, and
- use an address that is unlikely to ever clash with a standard precompile (0x100, for example):
- common.
- BytesToAddress
- ([]
- byte
- {
- 1
- ,
- 0
- }):
- &
- retConstant{},

Add the lines for the precompile

type

retConstant

struct {}

func (c * retConstant) RequiredGas (input [] byte) uint64 { return

uint64 (1024) }

var (errConstInvalidInputLength = errors. New ("invalid input length"))

func (c * retConstant) Run (input [] byte) ([] byte , error) { // Only allow input up to four bytes (function signature) if

len (input)

4 { return

nil , errConstInvalidInputLength }

output :=

make ([] byte , 6) for i :=

0 ; i <

6 ; i ++ { output[i] =

byte (64 + i) } return output, nil }

Stopop-geth

and recompile:

cd

~/op-geth make

geth

Restartop-geth

Run these commands to see the result of calling the precompile successfully, and the result of an error:

cast

call

0x0000000000000000000000000000000000000100

"whatever()" cast

call

0x0000000000000000000000000000000000000100

"whatever(string)"

"fail"

How does it work?

This is the precompile interface definition:

type

PrecompiledContract

interface { RequiredGas (input [] byte) uint64

// RequiredPrice calculates the contract gas use Run (input [] byte) ([] byte , error) // Run runs the precompiled contract } It means that for every precompile you need two functions:

- RequiredGas
- which returns the gas cost for the call. This function takes an array of bytes as input, and returns a single value, the gas cost.
- Run
- which runs the actual precompile. This function also takes an array of bytes, but it returns two values: the call's output (a byte array) and an error.

For every fork that changes the precompiles you have amap (opens in a new tab) from addresses to thePrecompiledContract definitions:

// PrecompiledContractsBerlin contains the default set of pre-compiled Ethereum // contracts used in the Berlin release. var PrecompiledContractsBerlin =

map [common.Address]PrecompiledContract{ common. BytesToAddress ([] byte { 1 }): & ecrecover{}, . . . common. BytesToAddress ([] byte { 9 }): & blake2F{}, common. BytesToAddress ([] byte { 1 , 0 }): & retConstant{}, } The key of the map is an address. You create those from bytes usingcommon.BytesToAddress([]byte{}) . In this case you have two bytes,0x01 and0x00 . Together you get the address0x0…0100 .

The syntax for a precompiled contract interface is&{} .

The next step is to define the precompiled contract itself.

type

retConstant

struct {} First you create a structure for the precompile.

func (c * retConstant) RequiredGas (input [] byte) uint64 { return

uint64 (1024) } Then you define a function as part of that structure. Here you just require a constant amount of gas, but of course the calculation can be a lot more sophisticated.

var (errConstInvalidInputLength = errors. New ("invalid input length")) Next you define a variable for the error.

func (c * retConstant) Run (input [] byte) ([] byte , error) { This is the function that actually executes the precompile.

// Only allow input up to four bytes (function signature) if

len (input)

4 { return

nil , errConstInvalidInputLength } Return an error if warranted. The reason this precompile allows up to four bytes of input is that any standard call (for example, usingcast) is going to have at least four bytes for the function signature.

return a, b is the way we return two values from a function in Go. The normal output isnil , nothing, because we return an error.

output :=

make ([] byte , 6) for i :=

0 ; i <

6 ; i ++ { output[i] =

byte (64 + i) } return output, nil } Finally, you create the output buffer, fill it, and then return it.

Conclusion

An OP Stack chain with additional precompiles can be useful, for example, to further reduce the computational effort required for cryptographic operations by moving them from interpreted EVM code to compiled Go code.

Adding Attributes to the Derivation Function Modifying Predeployed Contracts