How to customize your Orbit chain's precompiles

PUBLIC PREVIEW, MAINNET READY Orbit chains are nowMainnet ready ! Note that Orbit is still apublic preview capability - the Orbit product and its supporting documentation may change significantly as we capture feedback from readers like you.

To provide feedback, click theRequest an update button at the top of this document,join the Arbitrum Discord , or reach out to our team directly by completingthis form . caution The guidance in this document will work only if you useeth_call to call the new precompiles. If you're calling from other contracts or adding non-view/pure methods, this approach will break block validation.

To support these additional use-cases, follow the instructions described inHow to customize your Orbit chain's behavior. There are four primary ways to customize your chain's precompiles:

1. Add new methods to an existingprecompile
2. .
3. Create a new precompile.
4. Define a new event.
5. Customize gas usage for a specific method.
6. Call and modify state.

Prerequisites

Clone the Nitro repository before you begin:

git clone --branch v2.2.5 < https://github.com/OffchainLabs/nitro.git

> cd nitro git submodule update --init --recursive --force

Option 1: Add new methods to an existing precompile

Using your favorite code editor, open an existing precompile from theprecompiles implementation directory,/precompiles . We'll useArbSys.go as an example. Open the corresponding Go implementation file (ArbSys.go) and add a simpleSayHi method:

```
func

( con * ArbSys )

SayHi ( c ctx , evm mech )

( string ,

error )

{ return

"hi" ,
```

nil } Then, open the corresponding Solidity interface file (ArbSys.sol) from theprecompiles interface directory,/src/precompiles , and add the required interface. Ensure that the method name on the interface matches the name of the function you introduced in the previous step,camelCased :

```
function

sayHi ( )

external

view

returns ( string
```

memory) ; Next, follow the steps inHow to customize your Orbit chain's behaviorto build a modified Arbitrum Nitro node docker image and run it.

info Note that the instructions provided inHow to run a full node will not work with your Orbit node. SeeCommand-line options (Orbit) for Orbit-specific CLI flags. Once your node is running, you can callArbSys.sol either directly usingcurl , or through Foundry'scast call .

Call your function directly usingcurl

curl Your_IP_Address:8547 \ -X POST \ -H "Content-Type: application/json"

\ --data '{"method":"eth_call","params":[{"from":null,"to":"0x0000000000000000000000000000000000000064","data":"0x0c49c36c"}, "latest"],"id":1,"jsonrpc":"2.0"}' You should see something like this:

{"jsonrpc":"2.0","id":1,"result":"0x00000000000000000000000000000000000000000000000000000000000000200000000000000000000000000000000000000000000000000000000000000002686900
0x6869 is the hex-encoded utf8 representation ofhi , which you'll see embedded in theresult hex string.

Call your function using Foundry'scast call

cast call 0x0000000000000000000000000000000000000064 "sayHi()(string)” You should see something like this:

hi

Option 2: Create a new precompile

First, navigate to theprecompiles implementation directory,/precompiles , and create a new precompile implementation file calledArbHi.go . We'll define a new method, and we'll give it an address:

```
package precompiles

// ArbGasInfo provides insight into the cost of using the rollup. type ArbHi struct

{ Address addr // 0x11a, for example }

func

( con * ArbHi )

SayHi ( c ctx , evm mech )

( string ,

error )

{ return

"hi" ,
```

nil } Then, updateprecompile.go to register the new precompile under thePrecompiles() method:

```
insert ( MakePrecompile ( templates . ArbHiMetaData ,

& ArbHi { Address :

hex ( "11a" ) } ) )
```

// 0x011a here is an example address Navigate to theprecompiles interface directory,/src/precompiles , createArbHi.sol , and add the required interface. Ensure that the method name on the interface matches the name of the function you introduced in the previous step,camelCased :

pragma

solidity

= 0.4.21

< 0.9.0 ;

/// @title Say hi. /// @notice just for test /// This custom contract will set on 0x000000000000000000000000000000000000011a since we set it in precompile.go. interface

ArbHi

{ function

sayHi ( )

external

view

returns (string

memory) ; } Next, follow the steps inHow to customize your Orbit chain's behaviorto build a modified Arbitrum Nitro node docker image and run it.

info Note that the instructions provided inHow to run a full node will not work with your Orbit node. SeeCommand-line options (Orbit) for Orbit-specific CLI flags. Once your node is running, you can callArbHi.sol either directly usingcurl , or through Foundry'scast call .

Call your function directly usingcurl

curl Your_IP_Address:8547 \ -X POST \ -H "Content-Type: application/json"

\ --data '{"method":"eth_call","params":[{"from":null,"to":"0x000000000000000000000000000000000000011a","data":"0x0c49c36c"}, "latest"],"id":1,"jsonrpc":"2.0"}' You should see something like this:

{"jsonrpc":"2.0","id":1,"result":"0x0000000000000000000000000000000000000000000000000000000000000020000000000000000000000000000000000000000000000000000000000000000268690000

Call your function using Foundry'scast call

cast call 0x000000000000000000000000000000000000011a "sayHi()(string)” You should see something like this:

hi

Option 3: Define a new event

We'll reuse theArbsys precompile from Option 1 above to demonstrate how to emit a simpleHi event from theSayHi method inArbSys.sol .

First, go to theprecompiles implementation directory, findArbSys.go , and edit theArbSys struct:

// ArbSys provides system-level functionality for interacting with L1 and understanding the call stack. type ArbSys struct

{ Address addr // 0x64 L2ToL1Tx func (ctx , mech , addr , addr , huge , huge , huge , huge , huge , huge ,

[ ] byte)

error L2ToL1TxGasCost func (addr , addr , huge , huge , huge , huge , huge , huge ,

[ ] byte)

(uint64 ,

error) SendMerkleUpdate func (ctx , mech , huge , bytes32 , huge)

error SendMerkleUpdateGasCost func (huge , bytes32 , huge)

(uint64 ,

error) InvalidBlockNumberError func (huge , huge)

error

// deprecated event L2ToL1Transaction func (ctx , mech , addr , addr , huge , huge , huge , huge , huge , huge , huge ,

[ ] byte)

error L2ToL1TransactionGasCost func (addr , addr , huge , huge , huge , huge , huge , huge , huge ,

[ ] byte)

(uint64 ,

error)

// Add your customize event here: Hi func (ctx , mech , addr)

error // This is needed and will tell you how much gas it will cost, the param is the same as your event but without the first two (ctx, mech), the return param is always (uint64, error) HiGasCost func (addr)

(uint64 ,

error) } Then add the event to theSayHi method:

func

(con * ArbSys)

SayHi (c ctx , evm mech)

(string ,

error)

{ err := con . Hi (c , evm , c . caller) return

"hi" , err } Now navigate to theprecompiles interface directory, openArbsys.sol , and add the required interface. Ensure that the event name on the interface matches the name of the function you introduced inArbSys struct in the previous step:

event

Hi (address caller) ; If you want toindex the parameter of the event (if you want to filter by that parameter in the future, for example), just addindexed to the Solidity interface:

event

Hi (address

indexed caller) ; Our function now emits an event, which means that when calling it, the state will change and a gas cost will be incurred. So we have to remove theview function behavior:

function

sayHi ( )

external

returns (string

memory) ; Next, build Nitro by following the instructions inHow to build Nitro locally . Note that if you've already built the Docker image, you still need run the last step to rebuild.

Run Nitro with the following command:

docker run --rm -it -v /some/local/dir/arbitrum:/home/user/.arbitrum -p 0.0 .0.0:8547:8547 -p 0.0 .0.0:8548:8548 offchainlabs/nitro-node:v2.3.2-064fa11 --parent-chain.connection.url = < YourParentChainUrl

--chain.id

< YourOrbitChainId

--http.api

net,web3,eth,debug --http.corsdomain

* --http.addr

0.0 .0.0 --http.vhosts = * info Note that the instructions provided inHow to run a full node will not work with your Orbit node. SeeCommand-line options (Orbit) for Orbit-specific CLI flags.

Send the transaction and get the transaction receipt

To send a transaction toArbSys , we need to include a gas cost, because the function is no longer a view/pure function:

cast send 0x0000000000000000000000000000000000000064 "sayHi()(string)" Calleth_getTransactionReceipt with the returned transaction hash result. You should see something like this:

{"jsonrpc":"2.0","id":1,"result":
{"blockHash":"Your_blockHash","blockNumber":"Your_blockNumber","contractAddress":null,"cumulativeGasUsed":"0x680b","effectiveGasPrice":"0x5f5e100","from":"Your_address","gasUsed":"0x680b","
[{"address":"0x0000000000000000000000000000000000000064","topics":
["0xa9378d5bd800fae4d5b8d4c6712b2b64e8ecc86fdc831cb51944000fc7c8ecfa","0x000000000000000000000000{Your_address}"],"data":"0x","blockNumber":"Your_blockNumber","transactionHash":"\
Note thelogs field within the transaction receipt:

"logs":[{ "address":"0x0000000000000000000000000000000000000064", "topics":["0xa9378d5bd800fae4d5b8d4c6712b2b64e8ecc86fdc831cb51944000fc7c8ecfa", "0x000000000000000000000000{Your_address}"], "data":"0x", "blockNumber":"0x40", "transactionHash":"{Your_txHash}", "transactionIndex":"0x1", "blockHash":"0x0b367d705002b3575db99354a0964c033f929f26f4442ed347e47ae43a8f28e4", "logIndex":"0x0", "removed":false }]

Option 4: Customize gas usage for a specific method

The above instructions demonstrate how you can define a new precompile function. However, if this new function is simply defined without performing gas collection within the function, your precompile will be vulnerable to Denial-of-Service (DOS) attacks. These attacks exploit the function by flooding it with excessive requests without bearing the computational cost.

To deter this type of attack, you can implement a gas collection mechanism within your precompile. The event itself doesn't need to specify the gas cost; the program will calculate the gas cost when the event's execution is initially triggered.

In addition to introducing gas costs where they don't exist, you can also customize gas costs where they're already being incurred. To demonstrate, consider theGetBalance method inArbInfo.go :

// GetBalance retrieves an account's balance func (con ArbInfo) GetBalance(c ctx, evm mech, account addr) (huge, error) { if err := c.Burn(params.BalanceGasEIP1884); err != nil { return nil, err } return evm.StateDB.GetBalance(account), nil } The purpose of this method is to retrieve the balance of an address. As defined inEIP1884 , the operation code (opcode) for obtaining the address balance has an associated gas cost of 700 gas. The function accounts for this cost by deducting the specified amount of gas, indicated by the protocol constantBalanceGasEIP1884 , which is set to700 , through the call toc.Burn(int64) .

To customize the gas cost, let's implement an alternative toGetBalance , calledGetBalanceCustom :

// GetBalance retrieves an account's balance func (con ArbInfo) GetBalanceCustom(c ctx, evm mech, account addr) (huge, error) { gasForBalanceCall := uint64(300) if err := c.Burn(gasForBalanceCall); err != nil { return evm.StateDB.GetBalance(account), err } return balance, nil } To register this new precompile method, refer to Option 1 above.

Next, build Nitro by following the instructions inHow to build Nitro locally . Note that if you've already built the Docker image, you still need run the last step to rebuild.

Run Nitro with the following command:

docker run --rm -it -v /some/local/dir/arbitrum:/home/user/.arbitrum -p 0.0 .0.0:8547:8547 -p 0.0 .0.0:8548:8548 offchainlabs/nitro-node:v2.3.2-064fa11 --parent-chain.connection.url = < YourParentChainUrl

--chain.id

< YourOrbitChainId

--http.api

net,web3,eth,debug --http.corsdomain

* --http.addr

0.0 .0.0 --http.vhosts = * info Note that the instructions provided inHow to run a full node will not work with your Orbit node. SeeCommand-line options (Orbit) for Orbit-specific CLI flags.

Send the transaction and get the transaction receipt

In order to obtain the gas used, we can use theeth_sendRawTransaction RPC method to test execution on the chain. First, call:

cast send 0x0000000000000000000000000000000000000065 "GetBalance()({Any_Address})" Then, call:

cast send 0x0000000000000000000000000000000000000065 "GetBalanceCustom()({Any_Address})" The two responses will look like this, respectively:

Result 1:

{ "jsonrpc":"2.0", "id":1, "result":{ "blockHash":"{Your_blockHash}", "blockNumber":"0x15", "contractAddress":null, "cumulativeGasUsed":"0x638f", "effectiveGasPrice":"0x5f5e100", "from":"{Your_address}", "gasUsed":"0x638f", "gasUsedForL1":"0x9f5", "l1BlockNumber":"0x979a02", "logs":[

],
"logsBloom":"0x000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
"status":"0x1", "to":"0x0000000000000000000000000000000000000065", "transactionHash":"{Your_txHash}", "transactionIndex":"0x1", "type":"0x2" } } Result 2:

{ "jsonrpc":"2.0", "id":1, "result":{ "blockHash":"{Your_blockHash}", "blockNumber":"0x16", "contractAddress":null, "cumulativeGasUsed":"0x61ff", "effectiveGasPrice":"0x5f5e100", "from":"

{Your_address}", "gasUsed":"0x61ff", "gasUsedForL1":"0x9f5", "l1BlockNumber":"0x979a08", "logs":[

],
"logsBloom":"0x00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
"status":"0x1", "to":"0x0000000000000000000000000000000000000065", "transactionHash":"{Your_txHash}", "transactionIndex":"0x1", "type":"0x2" } } Here we can see that the gas cost incurred by the execution of the first transaction isgasUsed - gasUsedForL1 = 22938 . Similarly, the gas cost incurred by the execution of the second transaction is22538 . If you subtract the two, the result is400 , as expected.

To learn more about the gas cost model, seehow to estimate gas .

Option 5: Call and modify state

In this example, we'll demonstrate how toread from andwrite to a precompile contract'sArbOS state .

First, open thearbosstate.go file and locate theArbosState structure. This is where ArbOS state is defined.

Define a state key calledmyNumber of typestorage.StorageBackedUint64 . You can find more types instorage.go :

type ArbosState struct { // Other states infraFeeAccount storage.StorageBackedAddress brotliCompressionLevel storage.StorageBackedUint64 // brotli compression level used for pricing backingStorage *storage.Storage Burner burn.Burner myNumber storage.StorageBackedUint64 // this is what we added } Next, define the offset of your newly added state (tip: add it to the end so it won't affect other states):

const (versionOffset Offset = iota upgradeVersionOffset upgradeTimestampOffset networkFeeAccountOffset chainIdOffset genesisBlockNumOffset infraFeeAccountOffset brotliCompressionLevelOffset myNumberOffset // define the offset of your new state here) Then, initialize the state under theOpenArbosState andInitializeArbosState methods:

OpenArbosState:

return &ArbosState{ // other states backingStorage.OpenStorageBackedAddress(uint64(infraFeeAccountOffset)), backingStorage.OpenStorageBackedUint64(uint64(brotliCompressionLevelOffset)), backingStorage, burner, backingStorage.OpenStorageBackedUint64(uint64(myNumberOffset)), // define your new state here }, nil InitializeArbosState:

_ = sto.SetUint64ByUint64(uint64(versionOffset), 1) // initialize to version 1; upgrade at end of this func if needed _ = sto.SetUint64ByUint64(uint64(upgradeVersionOffset), 0) _ = sto.SetUint64ByUint64(uint64(upgradeTimestampOffset), 0) _ = sto.SetUint64ByUint64(uint64(myNumberOffset), 0) // initialize your new state around here Next, define your getter and setter::

func (state *ArbosState) SetNewMyNumber(newNumber uint64,) error { return state.myNumber.Set(newNumber) }

func (state *ArbosState) GetMyNumber() (uint64, error) { return state.myNumber.Get() } Next, head back to theprecompiles directory and create a newArbHi.go (introduced in Option 2). This time, we'll add two new methods to read and write the ArbOS state:

package precompiles

// ArbGasInfo provides insight into the cost of using the rollup. type ArbHi struct { Address addr // 0x11a, for example }

func (con *ArbHi) SayHi(c ctx, evm mech) (string, error) { return "hi", nil }

func (con *ArbHi) GetNumber(c ctx, evm mech) (uint64, error) { return c.State.GetMyNumber() }

func (con *ArbHi) SetNumber(c ctx, evm mech, newNumber uint64) error { return c.State.SetNewMyNumber(newNumber) } Follow the procedure detailed in Option 2 in order to add this new precompile contract, and then run your node.

Your smart contract interface should look like this:

pragma solidity >=0.4.21 <0.9.0;

/// @title Say hi. /// @notice just for test /// This custom contract will set on 0x000000000000000000000000000000000000011a since we set it in precompile.go. interface ArbHi { function sayHi() external view returns(string memory); function getNumber() external view returns(uint64); function setNumber(uint64) external; }

Send the transaction and get the transaction receipt

To send a transaction toArbSys , we need to include a gas cost, because the function is no longer a view/pure function:

cast send 0x000000000000000000000000000000000000011a "setNumber()" "2"

Get results from foundry cast

cast call 0x000000000000000000000000000000000000011a "getNumber()(uint64)” You should see something like this:

2 Edit this page Last updatedonMar 7, 2024 Previous How to customize your Orbit chain's behaviorNext Upgrade ArbOS