Geth at the core

The second key design idea in Nitro is "Geth at the core." Here "geth" refers to go-ethereum, the most common node software for Ethereum. As its name would suggest, go-ethereum is written in the Go programming language, as is almost all of Nitro.

The software that makes up a Nitro node can be thought of as built in three main layers, which are shown above:

- The base layer is the core of geth--the parts of Geth that emulate the execution of EVM contracts and maintain the data structures that make up the Ethereum state. Nitro compiles in this code as a library, with a few minor modifications to add necessary hooks.
- The middle layer, which we callArbOS
- , is custom software that provides additional functions associated with Layer 2 functionality, such as decompressing and parsing theSequencer
- 's data batches, accounting for Layer 1 gas costs and collecting fees to reimburse for them, and supporting cross-chainBridge
- functionalities such as deposits of Ether and tokens from L1 and withdrawals of the same back to L1. We'll dig in to the details of ArbOS below.
- The top layer consists of node software, mostly drawn from geth. This handles connections and incoming RPC requests from clients and provides the other top-level functionality required to operate an Ethereum-compatibleBlockchain
- node.

Because the top and bottom layers rely heavily on code from geth, this structure has been dubbed a "geth sandwich." Strictly speaking, Geth plays the role of the bread in the sandwich, and ArbOS is the filling, but this sandwich is named for the bread.

TheState Transition Function consists of the bottom Geth layer, and a portion of the middle ArbOS layer. In particular, the STF is a designated function in the source code, and implicitly includes all of the code called by that function. The STF takes as input the bytes of aTransaction received in the inbox, and has access to a modifiable copy of the Ethereum state tree. Executing the STF may modify the state, and at the end will emit the header of a new block (in Ethereum's block header format) which will be appended to the Nitro chain.

The rest of this section will be a deep dive into Geth and ArbOS. If deep technical knowledge does not suit you, skip to the next sectionSeparating Execution from Proving .

Geth

Nitro makes minimal modifications to Geth in hopes of not violating its assumptions. This section will explore the relationship between Geth and ArbOS, which consists of a series of hooks, interface implementations, and strategic re-appropriations of Geth's basic types.

We store ArbOS's state at an address inside a Gethstatedb . In doing so, ArbOS inherits thestatedb 's statefulness and lifetime properties. For example, a transaction's direct state changes to ArbOS are discarded upon a revert.

0xA4B05FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF

The fictional account representing ArbOS

info Please note any links on this page may be referencing old releases of Nitro or our fork of Geth. While we try to keep this up to date and most of this should be stable, please check against latest releases forNitro andGeth for most recent changes.

Hooks

Arbitrum uses various hooks to modify Geth's behavior when processing transactions. Each provides an opportunity for ArbOS to update its state and make decisions about the transaction during its lifetime. Transactions are applied using Geth'sApplyTransaction function.

Below isApplyTransaction 's callgraph, with additional info on where the various Arbitrum-specific hooks are inserted. Click on any to go to their section. By default, these hooks do nothing so as to leave Geth's default behavior unchanged, but for chains configured withEnableArbOS set to true,ReadyEVMForL2 installs the alternative L2 hooks.

- core.ApplyTransaction
- ➡core.applyTransaction
- ➡core.ApplyMessage
-
    - core.NewStateTransition
-

  -
    - ReadyEVMForL2
-
  - core.TransitionDb
-
  -
    - StartTxHook
-
  -
    - core.transitionDbImpl
-
  -
    -
      - ifIsArbitrum()
-
  -
    -
      - remove tip
-
  -
    -
      - GasChargingHook
-
  -
    -
      - evm.Call
-
  -
    -
      -
        - core.vm.EVMInterpreter.Run
-
  -
    -
      -
        -
          - PushCaller
-
  -
    -
      -
        -
          - PopCaller
-
  -
    -
      - core.StateTransition.refundGas
-
  -
    -
      -
        - ForceRefundGas
-
  -
    -
      -
        - NonrefundableGas
-
  -
    - EndTxHook
-
  - added return parameter:transactionResult

What follows is an overview of each hook, in chronological order.

ReadyEVMForL2

A call to ReadyEVMForL2 installs the other transaction-specific hooks into each Geth EVM right before it performs a state transition. Without this call, the state transition will instead use the default DefaultTxProcessor and get exactly the same results as vanilla Geth. A TxProcessor object is what carries these hooks and the associated Arbitrum-specific state during the transaction's lifetime.

StartTxHook

The StartTxHook is called by Geth before a transaction starts executing. This allows ArbOS to handle two Arbitrum-specific transaction types.

If the transaction is ArbitrumDepositTx , ArbOS adds balance to the destination account. This is safe because the L1 bridge submits such a transaction only after collecting the same amount of funds on L1.

If the transaction is an ArbitrumSubmitRetryableTx , ArbOS creates a retryable based on the transaction's fields. If the transaction includes sufficient gas, ArbOS schedules a retry of the new retryable.

The hook returns true for both of these transaction types, signifying that the state transition is complete.

GasChargingHook

This fallible hook ensures the user has enough funds to pay their poster's L1 calldata costs. If not, the transaction is reverted and the EVM does not start. In the common case that the user can pay, the amount paid for calldata is set aside for later reimbursement of the poster. All other fees go to the network account, as they represent the transaction's burden on validators and nodes more generally.

If the user attempts to purchase compute gas in excess of ArbOS's per-block gas limit, the difference is set aside and refunded later via ForceRefundGas so that only the gas limit is used. Note that the limit observed may not be the same as that seen at the start of the block if ArbOS's larger gas pool falls below the MaxPerBlockGasLimit while processing the block's previous transactions.

PushCaller

These hooks track the callers within the EVM callstack, pushing and popping as calls are made and complete. This provides ArbSys with info about the callstack, which it uses to implement the methods WasMyCallersAddressAliased and MyCallersAddressWithoutAliasing .

L1BlockHash

In Arbitrum, the BlockHash and Number operations return data that relies on underlying L1 blocks instead of L2 blocks, to accommodate the normal use-case of these opcodes, which often assume Ethereum-like time passes between different blocks. The L1BlockHash and L1BlockNumber hooks have the required data for these operations.

ForceRefundGas

This hook allows ArbOS to add additional refunds to the user's tx. This is currently only used to refund any compute gas purchased in excess of ArbOS's per-block gas limit during the GasChargingHook .

NonrefundableGas

Because poster costs come at the expense of L1 aggregators and not the network more broadly, the amounts paid for L1 calldata should not be refunded. This hook provides Geth access to the equivalent amount of L2 gas the poster's cost equals, ensuring this amount is not reimbursed for network-incentivized behaviors like freeing storage slots.

EndTxHook

The EndTxHook is called after the EVM has returned a transaction's result, allowing one last opportunity for ArbOS to intervene before the state transition is finalized. Final gas amounts are known at this point, enabling ArbOS to credit the network and poster each's share of the user's gas expenditures as well as adjust the pools. The hook returns from the TxProcessor a final time, in effect discarding its state as the system moves on to the next transaction where the hook's contents will be set afresh.

Interfaces and components

APIBackend

APIBackend implements the ethapi.Backend interface, which allows simple integration of the Arbitrum chain to existing Geth API. Most calls are answered using the Backend member.

Backend

This struct was created as an Arbitrum equivalent to the Ethereum struct. It is mostly glue logic, including a pointer to the ArbInterface interface.

ArbInterface

This interface is the main interaction-point between geth-standard APIs and the Arbitrum chain. Geth APIs mostly either check status by working on the Blockchain struct retrieved from the Blockchain call, or send transactions to Arbitrum using the PublishTransactions call.

RecordingKV

RecordingKV is a read-only key-value store, which retrieves values from an internal trie database. All values accessed by a RecordingKV are also recorded internally. This is used to record all preimages accessed during block creation, which will be needed to prove execution of this particular block. A RecordingChainContext should also be used, to record which block headers the block execution reads (another option would be to always assume the last 256 block headers were accessed). The process is simplified using two functions: PrepareRecording creates a stateDB and chaincontext objects, running block creation process using these objects records the required preimages, and PreimagesFromRecording function extracts the preimages recorded.

Transaction Types

Nitro Geth includes a few L2-specific transaction types. Click on any to jump to their section.

Tx Type Represents Last Hook Reached Source ArbitrumUnsignedTx An L1 to L2 message EndTxHook Bridge ArbitrumContractTx A nonce-less L1 to L2 message EndTxHook Bridge ArbitrumDepositTx A user deposit StartTxHook Bridge ArbitrumSubmitRetryableTx Creating a retryable StartTxHook Bridge ArbitrumRetryTx A Retryable Redeem attempt EndTxHook L2 ArbitrumInternalTx ArbOS state update StartTxHook ArbOS The following reference documents each type.

ArbitrumUnsignedTx

Provides a mechanism for a user on L1 to message a contract on L2. This uses the bridge for authentication rather than requiring the user's signature. Note, the user's acting address will be remapped on L2 to distinguish them from a normal L2 caller.

ArbitrumContractTx

These are like an ArbitrumUnsignedTx but are intended for smart contracts. These use the bridge's unique, sequential nonce rather than requiring the caller specify their own. An L1 contract may still use an ArbitrumUnsignedTx , but doing so may necessitate tracking the nonce in L1 state.

ArbitrumDepositTx

Represents a user deposit from L1 to L2. This increases the user's balance by the amount deposited on L1.

ArbitrumSubmitRetryableTx

Represents a retryable submission and may schedule anArbitrumRetryTx if provided enough gas. Please see theretryables documentation for more info.

ArbitrumRetryTx

These are scheduled by calls to theredeem method of theArbRetryableTx precompile and via retryable auto-redemption. Please see theretryables documentation for more info.

ArbitrumInternalTx

Because tracing support requires ArbOS's state-changes happen inside a transaction, ArbOS may create a transaction of this type to update its state in-between user-generated transactions. Such a transaction has aType field signifying the state it will update, though currently this is just future-proofing as there's only one value it may have. Below are the internal transaction types.

InternalTxStartBlock

Updates the L1 block number and L1 base fee. This transactionis generated whenever a new block is created. They areguaranteed to be the first in theirL2 Block .

Transaction Run Modes and Underlying Transactions

Ageth message may be processed for various purposes. For example, a message may be used to estimate the gas of a contract call, whereas another may perform the corresponding state transition. Nitro Geth denotes the intent behind a message by means of itsTxRunMode ,which it sets before processing it. ArbOS uses this info to make decisions about the transaction the message ultimately constructs.

A messagederived from a transaction will carry that transaction in a field accessible via itsUnderlyingTransaction method. While this is related to the way a given message is used, they are not one-to-one. The table below shows the various run modes and whether each could have an underlying transaction.

Run Mode Scope Carries an Underlying Tx? MessageCommitMode state transition always MessageGasEstimationMode gas estimation when created viaNodeInterface or when scheduled MessageEthcallMode eth_calls never

Arbitrum Chain Parameters

Nitro's Geth may be configured with the followingl2-specific chain parameters . These allow the rollup creator to customize their rollup at genesis.

EnableArbos

IntroducesArbOS , converting what would otherwise be a vanilla L1 chain into an L2 Arbitrum rollup.

AllowDebugPrecompiles

Allows access to debug precompiles. Not enabled forArbitrum One . When false, calls to debug precompiles will always revert.

DataAvailabilityCommittee

Currently does nothing besides indicate that the rollup will access a data availability service for preimage resolution in the future. This is not enabled for Arbitrum One, which is a strict state-function of its L1 inbox messages.

Miscellaneous Geth Changes

ABI Gas Margin

Vanilla Geth's abi library submits txes with the exact estimate the node returns, employing no padding. This means a transaction may revert should another arriving just before even slightly change the transaction's codepath. To account for this, we've added aGasMargin field tobind.TransactOpts thatpads estimates by the number of basis points set.

Conservation of L2 ETH

The total amount of L2 ether in the system should not change except in controlled cases, such as when bridging. As a safety precaution, ArbOS checks Geth'sbalance delta each time a block is created,alerting or panicking should conservation be violated.

MixDigest and ExtraData

To aid with[Outbox](https://docs.arbitrum.io/welcome/get-started) proof construction , the root hash and leaf count of ArbOS'ssend merkle accumulator are stored in theMixDigest andExtraData fields of each L2 block. The yellow paper specifies that theExtraData field may be no larger than 32 bytes, so we use the first 8 bytes of theMixDigest , which has no meaning in a system without miners/stakers, to store the send count.

Retryable Support

Retryables are mostly implemented inArbOS . Some modifications were required in Geth to support them.

- Added ScheduledTxes field to ExecutionResult. This lists transactions scheduled during the execution. To enable using this field, we also pass the ExecutionResult to callers of ApplyTransaction.
- Added gasEstimation param to DoCall. When enabled, DoCall will also also executing any retryable activated by the original call. This allows estimating gas to enable retryables.

Added accessors

AddedUnderlyingTransaction to Message interface AddedGetCurrentTxLogs to StateDB We created the AdvancedPrecompile interface, which executes and charges gas with the same function call. This is used byArbitrum's precompiles , and also wraps Geth's standard precompiles.

WASM build support

TheWASM Arbitrum executable does not support file operations. We createdfileutil.go to wrap fileutil calls, stubbing them out when building WASM.fake_leveldb.go is a similar WASM-mock for leveldb. These are not required for the WASM block-replayer.

Types

Arbitrum introduces a newsigner , and multiple newtransaction types .

ReorgToOldBlock

Geth natively only allows reorgs to a fork of the currently-known network. In nitro, reorgs can sometimes be detected before computing the forked block. We added theReorgToOldBlock function to support reorging to a block that's an ancestor of current head.

Genesis block creation

Genesis block in nitro is not necessarily block #0. Nitro supports importing blocks that take place before genesis. We split outWriteHeadBlock from genesis.Commit and use it to commit non-zero genesis blocks.

ArbOS

ArbOS is the Layer 2 EVM hypervisor that facilitates the execution environment of L2 Arbitrum. ArbOS is a trusted "system glue" component that runs at Layer 2 as part of the State Transition Function, it accounts for and manages network resources, produces blocks from incoming messages, cross-chain messaging, and operates its instrumented instance of Geth forSmart Contract execution.

In Arbitrum, much of the work that would otherwise have to be done expensively at Layer 1 is instead done by ArbOS, trustlessly performing these functions at the speed and low cost of Layer 2.

Supporting these functions in Layer 2 trusted software, rather than building them in to the L1-enforced rules of the architecture as Ethereum does, offers significant advantages in cost because these operations can benefit from the lower cost of computation and storage at Layer 2, instead of having to manage those resources as part of a Layer 1 contract. Having a trusted operating system at Layer 2 also has significant advantages in flexibility, because Layer 2 code is easier to evolve, or to customize for a particular chain, than a Layer-1 enforced architecture would be.

Precompiles

ArbOS provides L2-specific precompiles with methods smart contracts can call the same way they can solidity functions. Visit theprecompiles conceptual page for more information about how these work, and theprecompiles reference page for a full reference of the precompiles available in Arbitrum chains.

A precompile consists of a solidity interface incontracts/src/precompiles/ and a corresponding Golang implementation inprecompiles/ . Using Geth's ABI generator,solgen/gen.go generatessolgen/go/precompilesgen/precompilesgen.go , which collects the ABI data of the precompiles. Theruntime installer uses this generated file to check the type safety of each precompile's implementer.

The installer uses runtime reflection to ensure each implementer has all the right methods and signatures. This includes restricting access to stateful objects like the EVM and statedb based on the declared purity. Additionally, the installer verifies and populates event function pointers to provide each precompile the ability to emit logs and know their gas costs. Additional configuration like restricting a precompile's methods to only be callable by chain owners is possible by adding precompile wrappers likeownerOnly anddebugOnly to theirinstallation entry .

The calling, dispatching, and recording of precompile methods are done via runtime reflection as well. This avoids any human error manually parsing and writing bytes could introduce, and uses Geth's stable APIs forpacking and unpacking values.

Each time a transaction calls a method of an L2-specific precompile, acall context is created to track and record the gas burnt. For convenience, it also provides access to the public fields of the underlyingTxProcessor . Because sub-transactions could revert without updates to this struct, theTxProcessor only makes public that which is safe, such as the amount of L1 calldata paid by the top level transaction.

Messages

AnL1IncomingMessage represents an incoming sequencer message. A message includes one or more user transactions depending on load, and is made into aunique L2 block . The L2 block may include additional system transactions added in while processing the message's user transactions, but ultimately the relationship is still bijective: for everyL1IncomingMessage there is an L2 block with a unique L2 block hash, and for every L2 block after chain initialization there was anL1IncomingMessage that made it. A sequencerBatch may contain more than oneL1IncomingMessage .

Retryables

A Retryable is a special message type for creating atomic L1 to L2 messages; for details, seeL1 To L2 Messaging .

ArbOS State

ArbOS's state is viewed and modified viaArbosState objects, which provide convenient abstractions for working with the underlying data of itsbackingStorage . The backing storage'skeyed subspace strategy makes possibleArbosState 's convenient getters and setters, minimizing the need to directly work with the specific keys and values of the underlying storage'sstateDB .

Because twoArbosState objects with the samebackingStorage contain and mutate the same underlying state, differentArbosState objects can provide different views of ArbOS's contentsBurner objects, which track gas usage while working with theArbosState , provide the internal mechanism for doing so. Some are read-only, causing transactions to revert withvm.ErrWriteProtection upon a mutating request. Others demand the caller have elevated privileges. While yet others dynamically charge users when doing stateful work. For safety the kind of view is chosen whenOpenArbosState() creates the object and may never change.

Much of ArbOS's state exists to facilitate itsprecompiles . The parts that aren't are detailed below.

arbosVersion

,upgradeVersion andupgradeTimestamp

ArbOS upgrades are scheduled to happen when finalizing the first block after the upgradeTimestamp .

blockhashes

This component maintains the last 256 L1 block hashes in a circular buffer. This allows the TxProcessor to implement the BLOCKHASH and NUMBER opcodes as well as support precompile methods that involve the outbox. To avoid changing ArbOS state outside of a transaction, blocks made from messages with a new L1 block number update this info during an InternalTxUpdateL1BlockNumber ArbitrumInternalTx that is included as the first transaction in the block.

l1PricingState

In addition to supporting the ArbAggregator precompile , the L1 pricing state provides tools for determining the L1 component of a transaction's gas costs. This part of the state tracks both the total amount of funds collected from transactions in L1 gas fees, as well as the funds spent by batch posters to post data batches on L1.

Based on this information, ArbOS maintains an L1 data fee, also tracked as part of this state, which determines how much transactions will be charged for L1 fees. ArbOS dynamically adjusts this value so that fees collected are approximately equal to batch posting costs, over time.

l2PricingState

The L2 pricing state tracks L2 resource usage to determine a reasonable L2 gas price. This process considers a variety of factors, including user demand, the state of Geth, and the computational Speed Limit . The primary mechanism for doing so consists of a pair of pools, one larger than the other, that drain as L2-specific resources are consumed and filled as time passes. L1-specific resources like L1calldata are not tracked by the pools, as they have little bearing on the actual work done by the network actors that the speed limit is meant to keep stable and synced.

While much of this state is accessible through the ArbGasInfo and ArbOwner precompiles, most changes are automatic and happen during block production and the transaction hooks . Each of an incoming message's transactions removes from the pool the L2 component of the gas it uses, and afterward the message's timestamp informs the pricing mechanism of the time that's passed as ArbOS finalizes the block .

ArbOS's larger gas pool determines the per-block gas limit, setting a dynamic upper limit on the amount of compute gas an L2 block may have. This limit is always enforced, though for the first transaction it's done in the GasChargingHook to avoid sharp decreases in the L1 gas price from over-inflating the compute component purchased to above the gas limit. This improves UX by allowing the first transaction to succeed rather than requiring a resubmission. Because the first transaction lowers the amount of space left in the block, subsequent transactions do not employ this strategy and may fail from such compute-component inflation. This is acceptable because such transactions are only present in cases where the system is under heavy load and the result is that the user's transaction is dropped without charges since the state transition fails early. Those trusting the sequencer can rely on the transaction being automatically resubmitted in such a scenario.

The reason we need a per-block gas limit is that Arbitrator WAVM execution is much slower than native transaction execution. This means that there can only be so much gas -- which roughly translates to wall-clock time -- in an L2 block. It also provides an opportunity for ArbOS to limit the size of blocks should demand continue to surge even as the price rises.

ArbOS's per-block gas limit is distinct from Geth's block limit, which ArbOS sets sufficiently high so as to never run out. This is safe since Geth's block limit exists to constrain the amount of work done per block, which ArbOS already does via its own per-block gas limit. Though it'll never run out, a block's transactions use the same Geth gas pool to maintain the invariant that the pool decreases monotonically after each tx. Block headers use the Geth block limit for internal consistency and to ensure gas estimation works. These are both distinct from the gasLeft variable, which ephemerally exists outside of global state to both keep L2 blocks from exceeding ArbOS's per-block gas limit and to deduct space in situations where the state transition failed or used negligible amounts of compute gas. ArbOS does not need to persist gasLeft because it is its pool that induces a revert and because transactions use the Geth block limit during EVM execution. Edit this page Last updated on Jan 27, 2025

Previous The Sequencer and Censorship Resistance Next Separating Execution from Proving