ArbOS

ArbOS is the Layer 2 EVM hypervisor that facilitates the execution environment of L2 Arbitrum. ArbOS accounts for and manages network resources, produces blocks from incoming messages, and operates its instrumented instance of Geth for smart contract execution.

Precompiles

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

A precompile consists of a solidity interface in contracts/src/precompiles/ and a corresponding Golang implementation in precompiles/ . Using Geth's ABI generator, solgen/gen.go generates solgen/go/precompilesgen/precompilesgen.go , which collects the ABI data of the precompiles. The runtime 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 like ownerOnly and debugOnly to their installation 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 for packing and unpacking values.

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

Messages

An L1IncomingMessage represents an incoming sequencer message. A message includes one or more user transactions depending on load, and is made into a unique 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 every L1IncomingMessage there is an L2 block with a unique L2 block hash, and for every L2 block after chain initialization there was an L1IncomingMessage that made it. A sequencer batch may contain more than one L1IncomingMessage .

Retryables

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

ArbOS State

ArbOS's state is viewed and modified via ArbosState objects, which provide convenient abstractions for working with the underlying data of its backingStorage . The backing storage's keyed subspace strategy makes possible ArbosState 's convenient getters and setters, minimizing the need to directly work with the specific keys and values of the underlying storage's stateDB .

Because two ArbosState objects with the same backingStorage contain and mutate the same underlying state, different ArbosState objects can provide different views of ArbOS's contents. Burner objects, which track gas usage while working with the ArbosState , provide the internal mechanism for doing so. Some are read-only, causing transactions to revert with vm.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 when OpenArbosState() creates the object and may never change.

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

arbosVersion

, upgradeVersion and upgradeTimestamp

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 Apr 29, 2024
Previous L2 to L1 messaging and the outbox Next Geth