0-user-navigation-page.md:

---
title: Welcome to Morph
lang: en-US
keywords: [morph,ethereum,rollup,layer2,validity proof,optimistic zk-rollup]
description: Upgrade your blockchain experience with Morph - the secure decentralized, cost0efficient, and high-performing optimistic zk-rollup solution. Try it now!

We’re thrilled to have you here, exploring the world of Morph. If you’re a developer looking for technical documentation, head over to our For Developers section. But if you’re here to understand what Morph is all about, you’re in the right place!

!userpage

:::tip
Beta Testnet Stage: Morph is currently in the beta testnet phase, offering you a preview of many upcoming features for you to try. We encourage you to explore and test as much as you can.
:::

What is Morph?

Morph is a cutting-edge Layer 2 solution built on Ethereum, combining the best of optimistic rollups and zk technology. This makes us scalable, secure, and perfect for everyday applications. Our mission is to build the first blockchain for consumers, where user-friendly applications integrate seamlessly into everyday life, becoming indispensable utilities. We prioritize enabling blockchain applications that enhance daily experiences over chasing trivial technical milestones. Our focus is on creating meaningful, practical solutions that transform blockchain technology into core aspects of daily life

Getting Started

To help you navigate through our resources, here’s a quick guide:

Introductory Concepts: Get a general understanding of Morph, including what makes it unique, our vision and mission, key concepts, and a look at our roadmap. This section will help you grasp the basics and understand the foundation of Morph.

Step-by-Step Guides: Follow these practical tutorials for setting up your wallet, using the faucet, bridging assets, and exploring the Morph ecosystem. These guides are perfect for hands-on learning and getting started with Morph.

Links and Tools: Access useful tools such as the Morph Holesky Explorer and the official bridge. This section provides all the necessary links and resources to interact with and explore the Morph network effectively.

Engage in Our Ecosystem

Want to skip most of the reading and get started with exploring more right away? These are the guides that will get you right into the action:

- Wallet Setup: Learn how to set up your wallet to interact with Morph.
- Using the Faucet: Get test tokens to start experimenting on our testnet.
- Bridging Assets: Understand how to bridge assets between Morph and Ethereum.
- Exploring Morph Zoo: Dive into the diverse range of dApps and projects in the Morph ecosystem.

- Morph Holesky Explorer: Use our explorer to track transactions and explore the blockchain.

Explore and Connect

We believe in building a vibrant community around Morph. Join us on Discord to engage with other users and follow us on Twitter to stay updated with our latest news and developments.

1-overview-of-morph.md:

---
title: Overview
lang: en-US
keywords: [morph,layer2,validity proof,optimistic zk-rollup]
description: Upgrade your blockchain experience with Morph - the secure decentralized, cost0efficient, and high-performing optimistic zk-rollup solution. Try it now!

Welcome to Morph, a blockchain for consumers. At the core of Morph is a revolutionary approach to Ethereum Layer 2 scalability, harnessing the power of rollup technology.

Our platform is uniquely designed to enhance the blockchain experience, making it more accessible, efficient, and user-friendly for both developers and consumers. We achieve these innovations through a unique combination of cutting-edge technologies.

What Makes Morph Special

Given that our platform is built on the philosophy of consumer-centric innovation, we believe blockchain technology has the transformative power to enhance users’ daily lives. To accomplish this vision, our infrastructure has three core technological components that function as the foundation of a consumer blockchain.

Decentralized Sequencer Network

Morph’s distribution of the sequencing role eliminates single points of failure, reduces transaction censorship, and prevents monopolies over Miner Extractable Value (MEV). Building on Morph means guaranteed high availability, resilience, and fairness in transaction processing.

Optimistic zkEVM Integration

Morph’s Optimistic zkEVM Integration combines the efficiency of optimistic rollups with the security of zero-knowledge proofs, using a novel method called Responsive Validity Proof (RVP). RVP reduces costs and shortens withdrawal periods while maintaining high security, enabling robust applications that don’t compromise on performance or security.

Modular Design

Morph's architecture is divided into three modules: Sequencer Network for consensus and execution, Optimistic zkEVM for state verification, and Rollup for data availability. Developers can build and maintain applications more efficiently, ensuring each component can be upgraded independently without disrupting the entire system.

Before you dive into a deeper exploration of our technology, understanding our Vision and Mission will offer a better idea of why Morph stands out in the industry.

2-the-technology-behind-morph.md:

---
title: The Technology Behind Morph
lang: en-US
keywords: [morph,layer2,validity proof,optimistic zk-rollup]
description: Upgrade your blockchain experience with Morph - the secure decentralized, cost0efficient, and high-performing optimistic zk-rollup solution. Try it now!

Decentralized Sequencer Network

Morph's Decentralized Sequencer Network is designed to enhance the security and reliability of the blockchain. Unlike traditional Layer 2 solutions that rely on a centralized sequencer, Morph employs a network of decentralized sequencers. This setup ensures that no single entity has control over the transaction sequencing process, thereby eliminating the risk of a single point of failure. If one sequencer fails or acts maliciously, the others can continue processing transactions, maintaining the system's integrity and uptime. This decentralization also prevents transaction censorship and ensures that no single entity can monopolize Miner Extractable Value (MEV), creating a fairer environment for all users.

This collaborative approach not only increases security but also improves the overall efficiency and reliability of the transaction processing system, making Morph a robust and resilient Layer 2 solution.

!Sequqencer Network

Visit Morph’s Decentralized Sequencer Network for a more comprehensive article.

Optimistic zkEVM Integration

Optimistic and Zero-Knowledge (ZK) rollups are two distinct approaches to scaling blockchain transactions on layer 2. Optimistic rollups simply assume all transactions are valid when submitting a batch for settlement on Ethereum. However, the validity of any transaction can be contested by entities known as challengers, by submitting proof of fraudulent activity. If the fraud-proof is successful, the incorrect transaction is rejected, ensuring security but at the cost of some potential delays and high gas fees associated with the challenge process.

ZK rollups, on the other hand, use cryptographic proofs to verify the validity of transactions before these are submitted for settlement. All batches have their own ZK proof, allowing quick verification on the main chain without needing to review all the data associated with each transaction (hence “zero-knowledge”). This offers immediate finality with higher security, but generating

these proofs is computationally intensive and costly.

Morph’s hybrid rollup combines the best of these two approaches. Initially, the system operates optimistically, assuming transactions are valid to allow for quick processing and low costs. When a transaction is contested within Morph’s challenge window, it’s the sequencer that is required to produce a ZK proof to validate the transaction. We call this approach Responsive Validity Proof (RVP). It comes with the following improvements:

- Efficiency and Speed: A typical 7-day challenge window can be shortened to 1-3 days (a challenger no longer needs the extra time to identify malicious submissions, create a proof, and engage in multiple rounds of challenge procedures).
- Reduced Costs: Employing ZK-proofs means that only minimal transaction information is retained, thereby significantly reducing the cost of L2 submissions. When no challenges arise, the cost of ZK-proof submission and verification can be ignored. RVP is therefore more cost-effective than both optimistic and ZK rollups.

!Sequqencer Network

Visit Responsive Validity Proof for a more comprehensive article.

Modular Design

At its core, Morph is constructed using a sophisticated modular design architecture. The platform is organized into three functional modules (Sequencer Network, Rollup, Optimistic zk-EVM), each defined by distinct roles that collaborate in various configurations to meet diverse requirements. Each role within these modules operates its specific components, maintaining functional independence. This modular structure not only fosters flexibility and adaptability but also bolsters the composability of the system. It enables an efficient and interactive ecosystem, supporting the varied operational needs of our platform.

!Sequqencer Network

Visit Morph’s Modular Design for a more comprehensive article.

3-where-should-i-start.md:

---
title: Where Should I Start?
lang: en-US
keywords: [morph,layer2,validity proof,optimistic zk-rollup]
description: Upgrade your blockchain experience with Morph - the secure decentralized, cost0efficient, and high-performing optimistic zk-rollup solution. Try it now!

For Developers: Dive Into Building On Morph

As a developer keen on harnessing the innovative capabilities of the Morph network, your journey begins in the "Build on Morph" section of our documentation. Here, you'll find all the resources you need – from comprehensive guides and tutorials to practical examples. These materials are designed to elucidate the core concepts and features of Morph, enabling you to integrate with our APIs, deploy sophisticated smart contracts, and utilize an array of tools and libraries. The "Build on Morph" guide is your gateway to crafting

powerful, decentralized applications that leverage the full potential of our network.

For Users: Embark On Your Morph Journey

If you're eager to explore the possibilities within the Morph network, the "Quick Start" section in our documentation is your ideal starting point. Tailored for ease of understanding, this guide walks you through the essentials – from setting up your wallet to engaging in transactions on the network. It's designed to introduce you to the world of asset transfers, decentralized finance, and the diverse range of dApps available on Morph. Follow the "Quick Start" guide for a smooth, user-friendly introduction to leveraging Morph's functionalities in your everyday activities.

For Researchers: Unraveling The Technical Depth Of Morph

For those intrigued by the technical intricacies of the Morph network, the "How Morph Works" section offers an in-depth exploration of its underlying architecture and mechanisms. This comprehensive resource is rich in technical details, providing an extensive understanding of Morph's innovative features and foundational principles. Whether you're delving into research or seeking a thorough grasp of blockchain technology, this section unveils the nuances and sophisticated engineering that define the Morph network.

No matter your background – developer, user, or researcher – our documentation is structured to offer you a customized pathway into the Morph ecosystem. Each guide is designed to help you leverage and understand our state-of-the-art technology and features, ensuring a rewarding experience on the Morph network.

4-morphs-architecture.md:

---
title: Morph's Architecture
lang: en-US
keywords: [morph,layer2,validity proof,optimistic zk-rollup]
description: Upgrade your blockchain experience with Morph - the secure decentralized, cost0efficient, and high-performing optimistic zk-rollup solution. Try it now!

:::tip

This overview offers a concise introduction to Morph’s rollup technology stack. For an in-depth understanding, please refer to the "How Morph Works" section of our documentation.

:::

!Archi

The Modular Approach in Layer 2

Traditionally, the concept of modularity has been applied to Layer 1 blockchains, segmenting them into distinct layers. At Morph, we've extended this modular philosophy to Layer 2, building our platform around this principle.

In a typical Layer 1 blockchain, the architecture consists of four major layers:
- Consensus: The mechanism through which network agreement is achieved.
- Execution: Where transaction processing and smart contract operations occur.
- Settlement: The process of finalizing transactions.
- Data Availability: Ensuring that necessary information is accessible for

validation.

In the context of Layer 2, Morph reinterprets these layers with unique functionalities:

- Consensus and Execution via Decentralized Sequencer Network: At Morph, these functions are integrated and handled by our decentralized sequencer network. Sequencers orchestrate, process, and achieve consensus on Layer 2 transactions, forming the primary interface for user interactions.

!Archi

- Settlement with Optimistic zkEVM: Settlement in Morph refers to the finalization of Layer 2 transactions at the Ethereum level. It involves the crucial step of validating Layer 2 states. Morph employs the optimistic zkEVM for this purpose, a hybrid approach blending the best of optimistic rollups and zk-rollups. Layer 2 states will be eventually finalized by a significantly shorter challenge period or if gets challenged, a corresponding zk-proof.

!Archi

- Data Availability through 'Rollup' Process: This entails transferring essential Layer 2 data to Ethereum. In Morph, this is achieved through the 'Rollup' process, where a batch submitter compiles blocks into batches and submits them as Layer 1 transactions on Ethereum.

!Archi

Independent yet Collaborative Functions
Each of these major functions operates independently, facilitating asynchronous tasks and switchable implementations:
1. Sequencer Network: Executes Layer 2 transactions and updates local state.
2. Rollup Module: Transforms Layer 2 blocks into batches for submission to Layer 1.
3. State Verification: Utilizes Layer 1 security to verify Layer 2 states under the optimistic zkEVM rules.

This modular architecture enhances flexibility, adaptability, and composability within the Morph ecosystem.

Diverse Roles

Morph’s architecture is further defined by five pivotal roles: Sequencers, Validators, Nodes, Provers, and Layer 1 (Ethereum). Each role carries specific responsibilities and utilizes distinct components to fulfill its function, contributing to the seamless operation of the network.

For a deeper understanding of Morph’s architecture, please visit our comprehensive Developer Docs.

5-morphs-vision-and-mission.md:

---
title: Morph's Vision and Mission
lang: en-US
keywords: [morph,layer2,validity proof,optimistic zk-rollup]
description: Upgrade your blockchain experience with Morph - the secure decentralized, cost0efficient, and high-performing optimistic zk-rollup

solution. Try it now!

Morph is redefining the blockchain landscape with a clear focus on the consumer. Our vision is rooted in the belief that blockchain technology should be an accessible, practical tool for daily life, rather than a complex concept reserved for experts. We aim to transform how individuals interact with blockchain technology, making it as integral and user-friendly as any essential service in the digital age.

We Seek to Bridge the Gap

Our mission is to build an ecosystem of on-chain consumer applications on a completely decentralized infrastructure. Morph was created as a more approachable and practical blockchain solution in response to the gap in existing Layer 2 offerings – a lack of focus on the end-user experience and practicality.

Why Morph?

The inspiration behind Morph is a fusion of dreams and precision. Our platform is named after Morpheus, the Greek god of dreams, symbolizing our ability to influence both the commonplace and the extraordinary. Morph also draws on the mathematical principle of morphology, representing transformations that maintain structure and integrity. The combination of these concepts represents our ability to bring revolutionary blockchain applications that focus on improving everyday life with precision and reliability.

Commitment to Our Community

At Morph, our commitment extends beyond technology. We prioritize:

- Consumer-Centric Innovation: Focusing on the needs and experiences of users, ensuring that our platform is intuitive, efficient, and beneficial for everyday use.
- Transparency and Trust: Building a community grounded in openness and mutual trust, where every step we take is communicated clearly and honestly.
- Collaborative Ecosystem: Encouraging active participation and feedback from our community, ensuring that Morph evolves in alignment with the needs and demands of its users.

6-roadmap.md:

---
title: Roadmap
lang: en-US
keywords: [morph,layer2,validity proof,optimistic zk-rollup]
description: Upgrade your blockchain experience with Morph - the secure decentralized, cost0efficient, and high-performing optimistic zk-rollup solution. Try it now!

:::tip
 The roadmap is subject to changes based on technological developments,

community feedback, and external factors.
:::

!1

Phase 1: Testnet (2024 H1)

Testnet Sepolia (2024 Q1)

- Optimistic zkEVM
- Decentralized Sequencers

Testnet Holesky (2024 Q2)

- EIP-4844 Integration
- zkEVM Upgrade

Testnet Holesky

Phase 2: Mainnet Launch (2024 H2)

Mainnet Beta 2024 Q3
- Morph Staking
- AI Ecosystem Support

Mainnet Update Q4
- Account Abstraction
- Layer 3 Support

7-faqs.md:

---
title: FAQs
lang: en-US
keywords: [morph,layer2,validity proof,optimistic zk-rollup]
description: Upgrade your blockchain experience with Morph - the secure decentralized, cost0efficient, and high-performing optimistic zk-rollup solution. Try it now!

What Kind of Rollup is Morph?

Morph uses a hybrid rollup model called "Optimistic zkEVM & RVP". This combines the strengths of both zkRollups and Optimistic Rollups, optimizing for efficiency, cost, and speed. This unique approach places Morph at the cutting edge of rollup technology.

What Sets Morph Apart from Other Rollups?

Morph stands out with its innovative features:

State Verification: The Optimistic zkEVM & RVP method enhances efficiency by combining zkRollups and Optimistic Rollups.

Efficiency and Cost Reduction: Morph ensures fast transaction execution and cost-effectiveness while maintaining decentralization.

Decentralized Sequencer Network: This pioneering network setup addresses security concerns and ensures robustness at a Layer 1 level.

Modular Architecture: Morph’s adaptable and composable architecture fosters a flexible and interactive ecosystem.

As a Solidity Developer, Will I Notice Differences Deploying on Morph Compared to Ethereum?

Deploying on Morph is very similar to deploying on Ethereum, thanks to its EVM compatibility. Ethereum applications can be migrated to Morph with minimal adjustments. For more detailed guidance, check out the Development Guides section.

8-key-concepts.md:

---
title: Key Concepts
lang: en-US
keywords: [morph,layer2,validity proof,optimistic zk-rollup]
description: Upgrade your blockchain experience with Morph - the secure decentralized, cost0efficient, and high-performing optimistic zk-rollup solution. Try it now!

Optimistic Rollups

Optimistic rollups are a Layer 2 scaling solution for blockchains that enhance transaction throughput and reduce costs by assuming transactions are valid and only verifying them if a challenge is raised. This method relies on a challenge period during which validators can dispute transactions they believe to be incorrect. If no disputes are raised, the transactions are considered final. Optimistic rollups significantly improve scalability while maintaining security, making them an efficient solution for handling a higher volume of transactions on blockchain networks.

Learn more about Optimistic Rollups

ZK Rollups

ZK rollups, or zero-knowledge rollups, are a Layer 2 scaling solution that uses cryptographic proofs to verify the validity of transactions off-chain before bundling them and submitting a proof to the main blockchain. Each batch of transactions is accompanied by a zero-knowledge proof, which ensures that all transactions within the batch are valid without revealing the underlying data. This method provides immediate finality and high security, as the main chain only needs to verify the proof rather than each individual transaction, significantly reducing the computational load and enhancing scalability.

Learn more about ZK Rollups

Sequencers

Sequencers are specialized nodes responsible for ordering and bundling transactions in Layer 2 scaling solutions like rollups. They play a crucial role in determining the sequence of transactions, creating blocks, and periodically committing these blocks to the main blockchain. In decentralized systems, multiple sequencers work together to enhance security and prevent single points of failure. By ensuring transactions are processed efficiently and securely, sequencers help maintain the integrity and performance of Layer 2 networks.

Fraud Proof

Fraud proof is a mechanism used in blockchain scaling solutions like optimistic rollups to ensure transaction validity. When a sequencer submits a batch of transactions, they are assumed to be valid unless contested. During a designated challenge period, any validator or network participant can submit a fraud proof if they detect an incorrect transaction. This proof involves verifying the transaction data and demonstrating the error to the main blockchain. If the fraud proof is validated, the incorrect transaction is rejected, ensuring the integrity and security of the network while minimizing computational costs.

Validity Proof

Validity proof is a cryptographic method used to ensure that transactions within a rollup are correct before they are finalized on the main blockchain. In systems like ZK rollups, each batch of transactions is accompanied by a validity proof that verifies the correctness of all transactions within the batch. This approach enhances security and efficiency by eliminating the need for individual transaction verification on the main chain, providing immediate finality and reducing computational overhead.

zkEVM

zkEVM, or Zero-Knowledge Ethereum Virtual Machine, is an advanced implementation of the Ethereum Virtual Machine that integrates zero-knowledge proofs to enhance scalability and security. By using zk proofs, zkEVM allows for the validation of transactions off-chain, ensuring that only valid state transitions are submitted to the main chain. This method provides high throughput and lower transaction costs while maintaining the security and trustlessness of Ethereum.

BLS Signatures

BLS (Boneh-Lynn-Shacham) signatures are a cryptographic technique used to aggregate multiple signatures into a single compact signature. This is particularly useful in blockchain networks for reducing the data size and improving the efficiency of transaction verification. BLS signatures enable multiple validators to sign a message collectively, resulting in a single signature that can be verified quickly and cost-effectively, enhancing the overall scalability of the network.

Data availability

Data availability refers to the assurance that all necessary data for verifying blockchain transactions is accessible and retrievable. In the context of rollups, ensuring data availability is crucial for maintaining the integrity and security of off-chain transactions. It guarantees that anyone can download and verify the data used in rollup proofs, preventing scenarios where transactions are finalized without the possibility of verification.

EIP - 4844

EIP-4844, also known as Proto-Danksharding, is an Ethereum Improvement Proposal aimed at introducing a new type of transaction that reduces data costs and improves scalability. It involves adding a new transaction format that can efficiently handle large amounts of data, laying the groundwork for future sharding implementations. This proposal enhances the network's ability to manage data more effectively, contributing to overall improvements in throughput and cost-efficiency.

Discover how EIP-4844 impacts Morph and other rollups.

1-optimistic-rollup.md:

0-developer-navigation-page.md:

---
title: Developer Docs
lang: en-US
keywords: [morph,ethereum,rollup,layer2,validity proof,optimistic zk-rollup]
description: Upgrade your blockchain experience with Morph - the secure decentralized, cost0efficient, and high-performing optimistic zk-rollup solution. Try it now!

If you’re ready to build applications on Morph, you're in the right place.

For those who are regular users or visiting for the first time, we recommend starting with our For Users section to explore introductory content such as definitions, our vision, key concepts, and more.

!devintro

What is Morph?

Morph is the first optimistic zkEVM Ethereum Layer 2 solution that is 100% EVM compatible. Building on Morph is just like building on Ethereum. If you’re experienced in Ethereum development, you'll find your existing code, tooling, and dependencies are fully compatible with Morph.
Getting Started

To help you get started, here’s a recommended navigation through our documentation:

Fundamental Concepts: Start here to learn the core components of Morph, including Morph’s Modular Design, the Decentralized Sequencer Network, and the Responsive Validity Proof system.

Advanced Concepts: Dive deeper into topics such as understanding transaction costs on Morph and differences between Morph and Ethereum.

Developer Guides: Access comprehensive guides on verifying smart contracts, running a Morph node, deploying contracts, and more.

Developer Resources: Find detailed API methods, contract addresses and other useful developer resources.

Engage in Our Developer Community

We're actively enhancing our network with more integrations and support infrastructure, progressing towards our Mainnet release. Join our growing developer community. You can find us on Discord or follow our progress on Twitter.

1-intro.md:

---
title: Developer Docs
lang: en-US
keywords: [morph,ethereum,rollup,layer2,validity proof,optimistic zk-rollup]
description: Upgrade your blockchain experience with Morph - the secure decentralized, cost0efficient, and high-performing optimistic zk-rollup solution. Try it now!

Welcome to Morph’s Developer Docs!

Are you eager to build applications on Morph but unsure where to start?

You're in the right place – we've got you covered!

!devintro

As the first optimistic zkEVM Ethereum Layer 2 solution, Morph is 100% EVM compatible.

Building on morph is just like building on Ethereum. If you’re experienced in Ethereum development, you'll find your existing code, tooling, and dependencies are fully compatible with Morph.

What's next

Depending on what do you need

- For foundational knowledge: Check out our development basics
- For handy step by step contract deployment tutorial

- If you're well-versed and need specific developer resources like contract addresses, they are readily available.

Engage in Our Developer Community

We're actively enhancing our network with more integrations and support infrastructure, progressing towards our Mainnet release.

Join our growing developer community. You can find us on Discord, join our discussion forum, or follow our progress on Twitter.

1-difference-between-morph-and-ethereum.md:

---
title: Difference between Morph and Ethereum
lang: en-US
keywords: [morph,ethereum,rollup,layer2,validity proof,optimistic zk-rollup]
description: Upgrade your blockchain experience with Morph - the secure decentralized, cost0efficient, and high-performing optimistic zk-rollup solution. Try it now!

There are several technical differences between Ethereum’s EVM and Morph's optimistic zkEVM.

We’ve compiled a list to help you understand these distinctions better.

:::tip
For most Solidity developers, these technical details won't significantly impact your development experience.
:::

EVM Opcodes

| Opcode | Solidity equivalent | Morph Behavior |

| ------------------------- | ------------------ |

------------------------- |
| BLOCKHASH | block.blockhash | Returns keccak(chainid \|\|
blocknumber) for the last 256 blocks. |
| COINBASE | block.coinbase | Returns the pre-deployed fee
vault contract address. See Contracts |
| DIFFICULTY / PREVRANDAO | block.difficulty | Returns 0.
|
| BASEFEE | block.basefee | Disabled. If the opcode is
encountered, the transaction will be reverted. |
| SELFDESTRUCT | selfdestruct | Disabled. If the opcode is
encountered, the transaction will be reverted. |

EVM Precompiles

The RIPEMD-160 (address 0x3) blake2f (address 0x9), and point evaluation (address 0x0a) precompiles are currently not supported. Calls to unsupported precompiled contracts will revert. We plan to enable these precompiles in future hard forks.

The modexp precompile is supported but only supports inputs of size less than or equal to 32 bytes (i.e. u256).

The ecPairing precompile is supported, but the number of points(sets, pairs) is limited to 4, instead of 6.

The other EVM precompiles are all supported: ecRecover, identity, ecAdd, ecMul.

Precompile Limits

Because of a bounded size of the zkEVM circuits, there is an upper limit on the number of calls that can be made for some precompiles. These transactions will not revert, but simply be skipped by the sequencer if they cannot fit into the space of the circuit.

Precompile / Opcode	Limit
keccak256	3157
ecRecover	119
modexp	23
ecAdd	50
ecMul	50
ecPairing	2

:::tip Dencun upgrade opcode not available

Opcodes from the Dencun upgrade are not yet available on Morph, including MCOPY, TSTORE, TLOAD, BLOBHASH and BLOBBASEFEE. Additionally, EIP-4788 for accessing the Beacon Chain block root is not supported. We recommend using shanghai as your EVM target and avoiding using a Solidity version higher than 0.8.23.

:::

State Account

Additional Fields

We added two fields in the current StateAccount object: PoseidonCodehash and CodeSize.

```
go
type StateAccount struct {
      Nonce    uint64
```

```
	Balance  big.Int
	Root     common.Hash // merkle root of the storage trie
	KeccakCodeHash []byte // still the Keccak codehash
	// added fields
	PoseidonCodeHash []byte // the Poseidon codehash
	CodeSize uint64
}
```

CodeHash

Related to this, we maintain two types of codehash for each contract bytecode: Keccak hash and Poseidon hash.

KeccakCodeHash is kept to maintain compatibility for EXTCODEHASH. PoseidonCodeHash is used for verifying the correctness of bytecodes loaded in the zkEVM, where Poseidon hashing is far more efficient.

CodeSize

When verifying EXTCODESIZE, it is expensive to load the whole contract data into the zkEVM. Instead, we store the contract size in storage during contract creation. This way, we do not need to load the code — a storage proof is sufficient to verify this opcode.

Block Time

:::tip Block Time Subject to Change

Blocks are produced every second, with an empty block generated if there are no transactions for 5 seconds. However, this frequency may change in the future.
:::

To compare, Ethereum has a block time of 12 seconds.

Reasons for Faster Block Time in Morph
User Experience:

- A faster, consistent block time provides quicker feedback, enhancing the user experience.

- Optimization: As we refine the zkEVM circuits in our testnets, we can achieve higher throughput than Ethereum, even with a smaller gas limit per block or batch.

Notice:
- TIMESTAMP will return the timestamp of the block. It will update every second.
- BLOCKNUMBER will return an actual block number. It will update every second. The one-to-one mapping between blocks and transactions will no longer apply.

<!--
We also introduce the concept of system transactions that are created by the op-node, and are used to execute deposits and update the L2's view of L1. They have the following attributes:

- Every block will contain at least one system transaction called the L1 attributes deposited transaction. It will always be the first transaction in the block.
- Some blocks will contain one or more user-deposited transactions.
- All system transactions have an EIP-2718-compatible transaction type of 0x7E.

- All system transactions are unsigned, and set their v, r, and s fields to null.

:::Warning Known Issue
Some Ethereum client libraries, such as Web3j, cannot parse the null signature fields described above. To work around this issue, you will need to manually filter out the system transactions before passing them to the library.
:::
-->

Future EIPs

Morph closely monitors emerging Ethereum Improvement Proposals (EIPs) and adopts them when suitable. For more specifics, join our community forum or Discord for discussions.

<!-- EVM Target version

To avoid unexpected behaviors in your contracts, we recommend using ‘london’ as the target version when compiling your smart contracts.

You can read in more details on Shanghai hard fork differences from London on the Ethereum Execution spec and how the new PUSH0 instruction impacts the Solidity compiler.
-->

Transaction Fees

2-development-setup.md:

---
title: Development Setup
lang: en-US
keywords: [morph,ethereum,rollup,layer2,validity proof,optimistic zk-rollup]
description: Upgrade your blockchain experience with Morph - the secure decentralized, cost0efficient, and high-performing optimistic zk-rollup solution. Try it now!

Start Developing on Morph

Developing on Morph is as straightforward as developing on Ethereum.

To deploy contracts onto a MorphL2 chain, simply set the RPC endpoint of your target MorphL2 chain and deploy using your preferred Ethereum development framework:

- Hardhat
- Foundry
- Brownie
- Alchemy
- QuickNode SDK

...it all just works!

Holesky Testnet:

Step 1: Network Configuration

Before you start, ensure you are connected to the following networks:

Network Name	Morph Holesky Testnet	Holesky Testnet
RPC URL	https://rpc-quicknode-holesky.morphl2.io	https://ethereum-holesky-rpc.publicnode.com/
Chain ID	2810	17000
Currency Symbol	ETH	ETH
Block Explorer URL	https://explorer-holesky.morphl2.io/	https://holesky.etherscan.io/

:::tip Websocket Connection

wss://rpc-quicknode-holesky.morphl2.io

:::

Tendermint Consensus Information

Tendermint RPC: https://rpc-consensus-holesky.morphl2.io

Tendermint RPC Documentation: https://docs.tendermint.com/v0.34/rpc/#/

Step 2: Set up your developing framework

Hardhat

Modify your Hardhat config file hardhat.config.ts to point at the Morph public RPC.

jsx
```
const config: HardhatUserConfig = {
  ...
  networks: {
    morphl2: {
      url: 'https://rpc-quicknode-holesky.morphl2.io',
      accounts:
        process.env.PRIVATEKEY !== undefined ? [process.env.PRIVATEKEY] : [],
      gasprice = 2000000000
    },
  },
};
```

Foundry

To deploy using Morph Public RPC, run:

jsx
```
forge create ... --rpc-url= --legacy
```

ethers.js

Setting up a Morph provider in an ethers script:

jsx
```
import { ethers } from 'ethers';

const provider = new ethers.providers.JsonRpcProvider(
  'https://rpc-quicknode-holesky.morphl2.io'
```

);

Step 3: Acquire Ether

To start building on Morph, you may need some testnet ETH. Use a faucet to acquire holesky Ether, then bridge the test Ethereum Ether to the Morph testnet.

Each faucet has its own rules and requirements, so you may need to try a few before finding one that works for you.

Holesky ETH faucet websites:

https://stakely.io/en/faucet/ethereum-holesky-testnet-eth

https://faucet.quicknode.com/ethereum/holesky

https://holesky-faucet.pk910.de/

https://cloud.google.com/application/web3/faucet/ethereum (needs a Google account)

We have our own website faucet that can claim ETH & USDT for you initial usage.

Morph also offers a Discord faucet to obtain Morph Holesky USDT & Morph Holesky ETH.

Once you receive ETH on Holesky, you should see it in your wallet on the Holesky Network. It may take a few seconds for them to appear, but you can check the status by looking for a transaction to your address on a Holesky Block Explorer.

3-bridge-between-morph-and-ethereum.md:

---
title: Bridge between Morph and Ethereum
lang: en-US
keywords: [morph,ethereum,rollup,layer2,validity proof,optimistic zk-rollup]
description: Upgrade your blockchain experience with Morph - the secure decentralized, cost0efficient, and high-performing optimistic zk-rollup solution. Try it now!

Please first review our documentation on Communication Between Morph and Ethereum for some required fundamental knowledge.

Deposit ETH and ERC20 Tokens from L1

The Gateway Router allows ETH and ERC20 token bridging from L1 to L2 using the depositETH and depositERC20 functions respectively. It is a permissionless bridge deployed on L1. Notice that ERC20 tokens will have a different address on L2, you can use the getL2ERC20Address function to query the new address.

:::tip
  depositETH and depositERC20 are payable functions, the amount of ETH sent to these functions will be used
  to pay for L2 fees. If the amount is not enough, the transaction will not be sent. All excess ETH will be sent back to
  the sender. 0.00001 ETH should be more than enough to process a token deposit.
:::

When bridging ERC20 tokens, you don’t have to worry about selecting the right Gateway. This is because the L1GatewayRouter will choose the correct underlying entry point to send the message:

- L1StandardERC20Gateway: This Gateway permits any ERC20 deposit and will be selected as the default by the L1GatewayRouter for an ERC20 token that doesn’t need custom logic on L2. On the very first token bridging, a new token will be created on L2 that implements the MorphStandardERC20. To bridge a token, call the depositERC20 function on the L1GatewayRouter.

<!---->
<!--
- L1CustomERC20Gateway: This Gateway will be selected by the L1GatewayRouter for tokens with custom logic. For an L1/L2 token pair to work on the Morph Custom ERC20 Bridge, the L2 token contract has to implement IMorphStandardERC20. Additionally, the token should grant mint or burn capability to the L2CustomERC20Gateway.
-->

All Gateway contracts will form the message and send it to the L1CrossDomainMessenger which can send arbitrary messages to L2. The L1CrossDomainMessenger passes the message to the L1MessageQueueWithGasPriceOracle. Any user can send messages directly to the Messenger to execute arbitrary data on L2.

This means they can execute any function on L2 from a transaction made on L1 via the bridge. Although an application could directly pass messages to existing token contracts, the Gateway abstracts the specifics and simplifies making transfers and calls.

When a new block gets created on L1, the Sequencer will detect the message on the L1MessageQueue, and submit the transaction to the L2 via the L2 node. Finally, the L2 node will pass the transaction to the L2CrossDomainMessenger contract for execution on L2.

Withdraw ETH and ERC20 tokens from L2

The L2 Gateway is very similar to the L1 Gateway. We can withdraw ETH and ERC20 tokens back to L1 using the withdrawETH and withdrawERC20 functions. The contract address is deployed on L2. We use the getL1ERC20Address to retrieve the token address on L1.

:::tip
  withdrawETH and withdrawERC20 are payable functions, and the amount of ETH sent to these functions will be used to pay for L1 fees. If the amount is not enough, the transaction will not be sent. All excess ETH will be sent back to the sender. Fees will depend on L1 activity but 0.005 ETH should be enough to process a token withdrawal.
:::

:::tip
  Ensure transactions won’t revert on L1 while sending from L2 There is no way to recover bridged ETH, tokens, or NFTs if your transaction reverts on L1. All assets are burned on Morph when the transaction is sent, and it's impossible to mint them again.
:::

Finalize your Withdrawal

Besides starting a withdrawal request on Morph, there is one additional step to do. You need to finalize your withdrawal on Ethereum.

This is because of Morph's optimistic zkEVM design, you can read the details

here:

To do this, first you need to make sure:

- The batch containing the withdraw transactions has passed the challenge period and is marked as finalized (meaning in the Rollupcontract, withdrawalRoots[batchDataStore[batchIndex].withdrawalRoot] = true).

Once confirmed, you can call our backend services interface:

/getProof?nonce=withdraw.index

to obtain all the information you need to finalize your withdraw, which include:

- Index: The position of the withdrawal transaction in the withdraw tree, or rank of your transaction among all the L2->L1 transactions.
- Leaf: The hash value of your withdraw transaction that stored in the tree.
- Proof: The merkel proof of your withdraw transaction.
- Root: The withdraw tree root.

you need to use the proveAndRelayMessage function of the L1CrossDomainMessenger contract.

After obtaining the above information, the finalization of the withdraw operation can be carried out by calling L1CrossDomainMessenger.proveAndRelayMessage().

The required parameters are

```
solidity
  address from,
  address to,
  uint256 value,
  uint256 nonce,
  bytes memory message,
  bytes32[32] calldata withdrawalProof,
  bytes32 withdrawalRoot
```

from, to, value, nonce, and message are the original content of the withdraw transaction, which can be obtained from the Event SentMessage included in the transaction initiated by the L2 layer withdraw.

withdrawalProof and withdrawalRoot can be obtained from the aforementioned backend API interface.

<!--

Creating an ERC20 token with custom logic on L2

If a token needs custom logic on L2, it will need to be bridged through an L1CustomERC20Gateway and L2CustomERC20Gateway respectively. The custom token on L2 will need to give permission to the Gateway to mint new tokens when a deposit occurs and to burn when tokens are withdrawn

The following interface is the IMorphStandardERC20 needed for deploying tokens compatible with the L2CustomERC20Gateway on L2.

```
solidity
interface IMorphStandardERC20 {
  /// @notice Return the address of Gateway the token belongs to.
```

```
  function gateway() external view returns (address);

  /// @notice Return the address of counterpart token.
  function counterpart() external view returns (address);

  /// @dev ERC677 Standard, see https://github.com/ethereum/EIPs/issues/677
  /// Defi can use this method to transfer L1/L2 token to L2/L1,
  /// and deposit to L2/L1 contract in one transaction
  function transferAndCall(address receiver, uint256 amount, bytes calldata
data) external returns (bool success);

  /// @notice Mint some token to recipient's account.
  /// @dev Gateway Utilities, only gateway contract can call
  /// @param to The address of recipient.
  /// @param amount The amount of token to mint.
  function mint(address to, uint256 amount) external;

  /// @notice Burn some token from account.
  /// @dev Gateway Utilities, only gateway contract can call
  /// @param from The address of account to burn token.
  /// @param amount The amount of token to mint.
  function burn(address from, uint256 amount) external;
}
```

Adding a Custom L2 ERC20 token to the Morph Bridge

Tokens can be bridged securely and permissionlessly through Gateway contracts deployed by any developer. However, Morph also manages an ERC20 Router and a Gateway where all tokens created by the community are welcome. Being part of the Morph-managed Gateway means you won't need to deploy the Gateway contracts, and your token will appear in the Morph frontend. To be part of the Morph Gateway, you must contact the Morph team to add the token to both L1 and L2 bridge contracts. To do so, follow the instructions on the token lists repository to add your new token to the official Morph frontend.

-->

:::tip Use the SDK

You can also try our SDK to interact with the bridge system by referring to our SDK Doc.

:::
Add your Token to the Official Bridge

Currently, our official bridge only supports certain pre-defined tokens to be bridged. If you want to bridge your own tokens, you need to manually add the token, and here is how to do it.

Add Tokens to the gateway through Morph Bridge Frontend

The easiest way to support your token is to manually add it on our official bridge frontend, you can simply do it with the following steps:

1. Click the token selection button on Morph Holesky Bridge

!tokenlist1

2. Input & Confirm your desire Ethereum token contract address under the custom token section

!tokenlist2

3. Get the Layer 2 token contract address and confirm it.

!tokenlist3

4. Now it is supported and you and other users can start to bridge it!

!tokenlist4

Add token support to the bridge frontend

By adding your token to the gateway, you and other users can bridge the token by inputting the token address.You need to raise a PR to our bridge repo if you want your token shown on the bridge frontend token list.

You can find how to do it in the morph list repo.

Keep in mind that:
- You need to add both your L1 & L2 token to the list.
- The L2 token contract address is obtained by adding your tokens through Morph bridge frontend.

Here is an example PR commit for your reference.

4-understand-transaction-cost-on-morph.md:

---
title: Understand Transaction Cost on Morph
lang: en-US
keywords: [morph,ethereum,rollup,layer2,validity proof,optimistic zk-rollup]
description: Upgrade your blockchain experience with Morph - the secure decentralized, cost0efficient, and high-performing optimistic zk-rollup solution. Try it now!

Transaction fees on Morph work similarly to fees on Ethereum. However, Layer 2 introduces some unique aspects. Morph's optimistic zkEVM makes these differences easy to understand and even easier to handle.

This page includes the formula for calculating the gas cost of transactions on Morph.
There are two kinds of costs for transactions on Morph: the L2 execution fee and the L1 data/security fee.

<!--
:::tip

The transaction fees are collected into the SequencerFeeVault contract balance. This contract also tracks the amount we’ve historically withdrawn to L1 using totalProcessed()(uint256).

The block producer receives no direct reward, and the COINBASE opcode returns the fee vault address.

:::
-->

The L2 execution fee

Like Ethereum, transactions on Morph incur gas costs for computation and storage usage.

Every L2 transaction will pay some execution fee, equal to the amount of gas used multiplied by the gas price of the transaction.

Morph supports EIP-1559 transaction type. The EIP-1559 pricing model, which comprises a base fee and a priority fee, contributes to a more predictable and stable transaction fee.

The formula is straightforward:

l2executionfee = l2gasprice l2gasused
l2gasprice = l2basefee + l2priorityfee

The amount of L2 gas used depends on the specific transaction. Due to EVM compatibility, gas usage on Morph is typically similar to Ethereum.

The L1 data fee

Morph transactions are also published to Ethereum, crucial to Morph’s security as it ensures all data needed to verify Morph's state is always publicly available on Ethereum.

Users must pay for the cost of submitting their transactions to Ethereum, known as the L1 data fee. This fee typically represents most of the total cost of a transaction on Morph.

Formula:

l1DataFee = (l1BaseFee commitScalar + l1BlobBaseFee txdatagas blobScalar) / rcfg.Precision

where txdatagas is

txdatagas = countzerobytes(txdata) 4 + countnonzerobytes(txdata) 16

And other parameters:

1. l1BaseFee：Layer1 base fee
2. commitScalar: a factor used to measure the gas cost for data commitment
3. l1BlobBaseFee: the blobBaseFee on L1
4. blobScalar: a factor used to measure the gas cost for one transaction to be stored in EIP-4844 blob

:::tip
You can read the parameter values from the GasPrice oracle contract. Morph has pre-deployed GasPriceOracle, accessible on Morph Holesky at GasPriceOracle.
:::

Transaction fees' effect on software development

Sending transactions

The process of sending a transaction on Morph is identical to sending a transaction on Ethereum.

When sending a transaction, you should provide a gas price that is greater than or equal to the current L2 gas price.

Like on Ethereum, you can query this gas price with the ethgasPrice RPC method.

Similarly, you should set your transaction gas limit in the same way that you would set it on Ethereum (e.g. via ethestimateGas).

Displaying fees to users

Many Ethereum applications display estimated fees to users by multiplying the gas price by the gas limit.

However, as discussed earlier, users on Morph are charged both an L2 execution fee and an L1 data fee.

As a result, you should display the sum of both of these fees to give users the most accurate estimate of the total cost of a transaction.

Estimating the L2 execution fee

You can estimate the L2 execution fee by multiplying the gas price by the gas limit, just like on Ethereum.

Estimating the L1 data fee

Estimating the total fee

You can estimate the total fee by combining your estimates for the L2 execution fee and L1 data fee.

Sending max ETH

Sending the maximum amount of ETH that a user has in their wallet is a relatively common use case.

When doing this, you will need to subtract the estimated L2 execution fee and the estimated L1 data fee from the amount of ETH you want the user to send.

Use the logic described above for estimating the total fee.

Common RPC Errors

Insufficient funds

- Error code: -32000
- Error message: invalid transaction: insufficient funds for l1Fee + l2Fee + value

You'll get this error when attempting to send a transaction and you don't have enough ETH to pay for the value of the transaction, the L2 execution fee, and the L1 data fee.
You might get this error when attempting to send max ETH if you aren't properly accounting for both the L2 execution fee and the L1 data fee.

Gas price too low

- Error code: -32000
- Error message: gas price too low: X wei, use at least tx.gasPrice = Y wei

This is a custom RPC error that Morph returns when a transaction is rejected because the gas price is too low.
See the section on Responding to gas price updates for more information.

Gas price too high
- Error code: -32000
- Error message: gas price too high: X wei, use at most tx.gasPrice = Y wei

This is a custom RPC error that Morph returns when a transaction is rejected because the gas price is too high.
We include this as a safety measure to prevent users from accidentally sending a transaction with an extremely high L2 gas price.
See the section on Responding to gas price updates for more information.

5-verify-your-smart-contracts.md:

---
title: Verify Your Smart Contracts
lang: en-US
keywords: [morph,ethereum,rollup,layer2,validity proof,optimistic zk-rollup]
description: Upgrade your blockchain experience with Morph - the secure decentralized, cost0efficient, and high-performing optimistic zk-rollup solution. Try it now!

After deploying your smart contracts, it's crucial to verify your code on our block explorer. This can be automated using your development framework, such as Hardhat.

Verify with development framework

Most smart contract tools have plugins for verifying contracts on Etherscan. Blockscout supports Etherscan's contract verification APIs, making it straightforward to use these tools with the Morph Testnet.

Verify with Hardhat

To verify your contract through hardhat, you need to add the following Etherscan and Sourcify configs to your hardhat.config.js file:

```
javascript
module.exports = {
  networks: {
    morphTestnet: { ... }
  },
  etherscan: {
    apiKey: {
      morphTestnet: 'anything',
    },
    customChains: [
      {
        network: 'morphTestnet',
        chainId: 2810,
        urls: {
          apiURL: 'https://explorer-api-holesky.morphl2.io/api? ',
          browserURL: 'https://explorer-holesky.morphl2.io/',
        },
```

```
      },
    ],
  },
};
```

Verify with Foundry

Verification with foundry requires some flags passed to the normal verification script. You can verify using the command below:

```
bash
 forge verify-contract YourContractAddress Counter\
  --chain 2810 \
  --verifier-url https://explorer-api-holesky.morphl2.io/api? \
  --verifier blockscout --watch
```

Verify with Morph explorer frontend

- Visit：Morph block explorer

We currently support 6 different ways to verify your contracts on our block explorer.

There are 2 general parameters:

- Compiler: Has to be consistent with what you select when deployment.
- Optimization: Can be ignored if you don't have contract optimization. If you do, it has to be consistent with deployment.

Method: Solidity (Flattened Sources Code)

Frontend:

!fscs

Flatten

Flatten through forge command, for example:

```
forge flatten --output FlattenedL2StandardBridge.sol
./contracts/L2/L2StandardBridge.sol
```

Method: Solidity (Standard JSON Input)
!sjis1

Obtain JSON File

- Can be obtained through solc
- Can be obatined through remix compiler

!sjis2

!sjis3
Method: Solidity (Multi-part files)

Frontend:

- You can submit multiple contract file by your own needs

!mpfs1

SOL File Process
- If there is any imported file, it needs to be modified to be referenced by the same level path, and these files must be submitted together.
!mpfs2
Method: Vyper (Contracts)

Frontend:
!cv
Method: Vyper (Standard Json Input)

Frontend:
!sjiv
Method: Vyper (Multi-part files)

Frontend:
!mpfv

After Verification

!avp

1-deploy-contract-on-morph.md:

---
title: Deploy Contracts on Morph
lang: en-US
keywords: [morph,ethereum,rollup,layer2,validity proof,optimistic zk-rollup]
description: Upgrade your blockchain experience with Morph - the secure decentralized, cost0efficient, and high-performing optimistic zk-rollup solution. Try it now!

The Morph Holesky Testnet allows anyone to deploy a smart contract on Morph. This tutorial will guide you through deploying a contract on Morph Holesky using common Ethereum development tools.

This demo repo illustrates contract deployment with Hardhat and Foundry.

:::tip
  Before you start deploying the contract, you need to request test tokens from a Holesky faucet and use the
  bridge to transfer some test ETH from Holesky to Morph Holesky.

  See our Faucet for details.
:::

<!--

Deploy contracts with Remix

-->

Deploy with Hardhat

Clone the repo

bash
git clone https://github.com/morph-l2/morph-examples.git

Install Dependencies

If you haven't already, install nodejs and yarn.

bash
cd contract-deployment-demos/hardhat-demo
yarn install

This will install everything you need include hardhat for you.

Compile

Compile your contract

bash
yarn compile

Test

This will run the test script in test/Lock.ts

bash
yarn test

Deploy

Create a .env file following the example .env.example in the root directory. Change PRIVATEKEY to your own account private key in the .env.

And Change the network settings in the hardhat.config.ts file with the following information:

javascript
```
morphTestnet: {
  url: process.env.MORPHTESTNETURL || "",
  accounts:
    process.env.PRIVATEKEY !== undefined ? [process.env.PRIVATEKEY] : [],
}
```

Then run the following command to deploy the contract on the Morph Holesky Testnet. This will run the deployment script that set the initialing parameters, you can edit the script in scripts/deploy.ts

bash
yarn deploy:morphTestnet

Verify your contracts on Morph Explorer

To verify your contract through hardhat, you need to add the following Etherscan and Sourcify configs to your hardhat.config.js file:

javascript
```
module.exports = {
  networks: {
    morphTestnet: { ... }
  },
```

```
  etherscan: {
    apiKey: {
      morphTestnet: 'anything',
    },
    customChains: [
      {
        network: 'morphTestnet',
        chainId: 2810,
        urls: {
          apiURL: 'https://explorer-api-holesky.morphl2.io/api? ',
          browserURL: 'https://explorer-holesky.morphl2.io/',
        },
      },
    ],
  },
};
```

Then run the hardhat verify command to finish the verification

```
bash
npx hardhat verify --network morphTestnet DEPLOYEDCONTRACTADDRESS
<ConstructorParameter>
```

For example

```
bash
npx hardhat verify --network morphTestnet
0x8025985e35f1bAFfd661717f66fC5a434417448E '0.00001'
```

Once succeed, you can check your contract and the deployment transaction on Morph Holesky Explorer

Deploy contracts with Foundry

Clone the repo

```
bash
git clone https://github.com/morph-l2/morph-examples.git
```

Install Foundry

```
bash
curl -L https://foundry.paradigm.xyz | bash
foundryup
```

Then go the right folder of our example:

```
bash
cd contract-deployment-demos/foundry-demo
```

Compile

```
bash
forge build
```

Deploy

A Deployment script and use of environment variables has already been set up for

you. You can view the script at script/Counter.s.sol

Rename your .env.example file to .env and fill in your private key. The RPC URL has already been filled in along with the verifier URL.

To use the variables in your .env file run the following command:

shell
source .env

You can now deploy to Morph with the following command:

shell
forge script script/Counter.s.sol --rpc-url $RPCURL --broadcast --private-key $DEPLOYERPRIVATEKEY --legacy

Adjust as needed for your own script names.

Verify

Verification requires some flags passed to the normal verification script. You can verify using the command below:

bash
 forge verify-contract YourContractAddress Counter\
  --chain 2810 \
  --verifier-url $VERIFIERURL \
  --verifier blockscout --watch

Once succeeded, you can check your contract and the deployment transaction on Morph Holesky Explorer.

Questions and Feedback

Thank you for participating in and developing on the Morph Holesky Testnet! If you encounter any issues, join our Discord and find us at #dev-support channel.

2-bridge-between-morph-and-ethereum.md:

---
title: Bridge between Morph and Ethereum
lang: en-US
keywords: [morph,ethereum,rollup,layer2,validity proof,optimistic zk-rollup]
description: Upgrade your blockchain experience with Morph - the secure decentralized, cost0efficient, and high-performing optimistic zk-rollup solution. Try it now!

Bridge an ERC20 through custom gateway

Step 1: Launch a token on Holesky

First, we need a token to bridge. There is no need for a particular ERC20 implementation in order for a token to be compatible with L2. If you already have a token, feel free to skip this step. If you want to deploy a new token, use the following contract of a simple ERC20 token that mints 1 million tokens

to the deployer when launched.

solidity
```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.16;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract L1Token is ERC20 {
  constructor() ERC20("My Token L1", "MTL1") {
    mint(msg.sender, 1000000 ether);
  }
}
```

Step 2: Launch the counterpart token on Morph Holesky testnet

Next, you'll launch a counterpart to this token on Morph, which will represent the original token on Holesky. This token can implement custom logic to match that of the L1 token or even add additional features beyond those of the L1 token.

For this to work:

- The token must implement the IMorphStandardERC20 interface in order to be compatible with the bridge.
- The contract should provide the gateway address and the counterpart token addresses (the L1 token we just launched) under the gateway() and counterpart() functions. It should also allow the L2 gateway to call the token mint() and burn() functions, which are called when a token is deposited and withdrawn.

The following is a complete example of a token compatible with the bridge. To the constructor, you will pass the official Morph Custom Gateway address (0x058dec71E53079F9ED053F3a0bBca877F6f3eAcf) and the address of the token launched on Holesky.

solidity
```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.16;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@Morph-tech/contracts@0.1.0/libraries/token/IMorphERC20Extension.sol";

contract L2Token is ERC20, IMorphERC20Extension {
  // We store the gateway and the L1 token address to provide the gateway() and counterpart() functions which are needed from the Morph Standard ERC20 interface
  address gateway;
  address counterpart;

  // In the constructor we pass as parameter the Custom L2 Gateway and the L1 token address as parameters
  constructor(address gateway, address counterpart) ERC20("My Token L2", "MTL2") {
    gateway = gateway;
    counterpart = counterpart;
  }

  function gateway() public view returns (address) {
    return gateway;
  }

  function counterpart() external view returns (address) {
    return counterpart;
  }
```

```
  // We allow minting only to the Gateway so it can mint new tokens when bridged
from L1
  function transferAndCall(address receiver, uint256 amount, bytes calldata
data) external returns (bool success) {
    transfer(receiver, amount);
    data;
    return true;
  }

  // We allow minting only to the Gateway so it can mint new tokens when bridged
from L1
  function mint(address to, uint256 amount) external onlyGateway {
    mint(to, amount);
  }

  // Similarly to minting, the Gateway is able to burn tokens when bridged from
L2 to L1
  function burn(address from, uint256 amount) external onlyGateway {
    burn(from, amount);
  }

  modifier onlyGateway() {
    require(gateway() == msgSender(), "Ownable: caller is not the gateway");
    ;
  }
}
```

Step 3: Add the token to the Morph Bridge

You need to contact the Morph team to add the token to L2CustomERC20Gateway contract in Morph and L1CustomERC20Gateway contract in L1. In addition, follow the instructions on the token lists repository to add your token to the Morph official bridge frontend.

Step 4: Deposit tokens

Once your token has been approved by the Morph team, you should be able to deposit tokens from L1. To do so, you must first approve the L1CustomGateway contract address on Holesky (0x31C994F2017E71b82fd4D8118F140c81215bbb37). Then, deposit tokens by calling the depositERC20 function from the L1CustomGateway contract. This can be done using our bridge UI, Etherscan Holesky, or a smart contract.

Step 5: Withdraw tokens

You will follow similar steps to send tokens back from L2 to L1. First, approve the L2CustomGateway address (0x058dec71E53079F9ED053F3a0bBca877F6f3eAcf) and then withdraw the tokens calling the withdrawERC20 from the L2CustomGateway contract.

Send messages between Morph and Ethereum

Deploying the Contracts

Target Smart Contract

Let’s start by deploying the target smart contract. We will use the Greeter contract for this
example, but you can use any other contract. Deploy it to either Holesky or Morph. On Morph, L1
and L2 use the same API, so it’s up to you.

```
solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.16;

// This Greeter contract will be interacted with through the MorphMessenger
across the bridge
contract Greeter {
  string public greeting = "Hello World!";

  // This function will be called by executeFunctionCrosschain on the Operator
Smart Contract
  function setGreeting(string memory greeting) public {
    greeting = greeting;
  }
}
```

We will now execute setGreeting in a cross-chain way.

Operator Smart Contract

Switch to the other chain and deploy the GreeterOperator. So, if you deployed the Greeter contract on L1, deploy the GreeterOperator on L2 or vice versa.

```
solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.16;

// The Morph Messenger interface is the same on both L1 and L2, it allows
sending cross-chain transactions
// Let's import it directly from the Morph Contracts library
import "@Morph-tech/contracts@0.1.0/libraries/IMorphMessenger.sol";

// The GreeterOperator is capable of executing the Greeter function through the
bridge
contract GreeterOperator {
  // This function will execute setGreeting on the Greeter contract
  function executeFunctionCrosschain(
    address MorphMessengerAddress,
    address targetAddress,
    uint256 value,
    string memory greeting,
    uint32 gasLimit
  ) public payable {
    IMorphMessenger MorphMessenger = IMorphMessenger(MorphMessengerAddress);
    // sendMessage is able to execute any function by encoding the abi using the
encodeWithSignature function
    MorphMessenger.sendMessage{ value: msg.value }(
      targetAddress,
      value,
      abi.encodeWithSignature("setGreeting(string)", greeting),
      gasLimit,
      msg.sender
    );
  }
}
```

Calling a Cross-chain Function

We pass the message by executing executeFunctionCrosschain and passing the following parameters:

- MorphMessengerAddress: This will depend on where you deployed the GreeterOperator contract.
  - If you deployed it on Holesky use 0x50c7d3e7f7c656493D1D76aaa1a836CedfCBB16A. If you deployed on Morph Holesky use 0xBa50f5340FB9F3Bd074bD638c9BE13eCB36E603d.
- targetAddress: The address of the Greeter contract on the opposite chain.
- value: In this case, it is 0 because the setGreetingis not payable.
- greeting: This is the parameter that will be sent through the message. Try passing “This message was cross-chain!”
- gasLimit:
  - If you are sending the message from L1 to L2, around 1000000 gas limit should be more than enough. That said, if you set this too high, and msg.value doesn't cover gasLimit baseFee, the transaction will revert. If msg.value is greater than the gas fee, the unused portion will be refunded.
  - If you are sending the message from L2 to L1, pass 0, as the transaction will be completed by executing an additional transaction on L1.

Relay the Message when sending from L2 to L1

When a transaction is passed from L2 to L1, an additional "execute withdrawal transaction" must be sent on L1. To do this, you must call relayMessageWithProof on the L1 Morph Messenger
contract from an EOA wallet.

You can do this directly on Etherscan Holesky.
To do so, you will need to pass a Merkle inclusion proof for the bridged transaction and other parameters. You'll query these using the Morph Bridge API.

{/ TODO: finish looking into API issues /}

We're finalizing the API specifics, but for now, fetch or curl the following endpoint:

bash
curl "https://Holesky-api-bridge.Morph.io/api/claimable?pagesize=10&page=1&address=GREETEROPERATORADDRESSONL2"

<!--
Replace GREETEROPERATORADDRESSONL2 with your GreeterOperator contract address as launched on L2. Read more about Execute Withdraw transactions
in the Morph Messenger article.
-->

:::tip
  This API was made for our Bridge UI. It is not yet finalized and may change in the future. We will update this guide
  when the API is finalized.
:::

:::tip Anyone can execute your message
  relayMessageWithProof is fully permissionless, so anyone can call it on your behalf if they're willing to pay the L1
  gas fees. This feature allows for additional support infrastructure, including tooling to automate this process for
  applications and users.
:::

After executing and confirming the transaction on both L1 and L2, the new state of greeting on the Greeter contract should be “This message was cross-chain!”. Sending a message from one chain to the other should take around 20 minutes after the transactions are confirmed on the origin chain.

Congratulations, you now executed a transaction from one chain to the other

using our native bridge!

1-contracts.md:

---
title: Contract Addresses
lang: en-US

:::info

Contract address could be dynamic during the public testnet stage, so it is recommended to visit this page frequently.

:::

Morph Holesky Network Info

Network Name	Morph Holesky Testnet	Ethereum Holesky Testnet
RPC URL	https://rpc-quicknode-holesky.morphl2.io	https://ethereum-holesky-rpc.publicnode.com/
Chain ID	2810	17000
Currency Symbol	ETH	ETH
Block Explorer URL	https://explorer-holesky.morphl2.io/	https://holesky.etherscan.io/

Morph Holesky Contracts

Main Contract

L1 Contract

Staking: 0x868dd5d1c268277e331b726bb438edde8221d389

Rollup: 0xd8c5c541d56f59d65cf775de928ccf4a47d4985c

L1MessageQueueWithGasPriceOracle: 0x778d1d9a4d8b6b9ade36d967a9ac19455ec3fd0b

L1CrossDomainMessenger: 0xecc966ab425f3f5bd58085ce4ebdbf81d829126f

L2 Contract

L2ToL1MessagePasser: 0x5300000000000000000000000000000000000001

L2Sequencer: 0x5300000000000000000000000000000000000003

L2Gov: 0x5300000000000000000000000000000000000004

L2Submitter: 0x5300000000000000000000000000000000000005

L2CrossDomainMessenger: 0x5300000000000000000000000000000000000007

Gateway

L1 Gateway

L1GatewayRouter: 0xea593b730d745fb5fe01b6d20e6603915252c6bf

L1ETHGateway: 0xcc3d455481967dc97346ef1771a112d7a14c8f12

L1WETHGateway: 0xbbdb317b50313d96823eba0fc2c1d9e469dc1906

L1StandardERC20Gateway: 0xb26dafdb434ae93e3b8efde4f0193934955d86cd

L2 Gateway

L2GatewayRouter: 0x5300000000000000000000000000000000000002

L2StandardERC20Gateway: 0x5300000000000000000000000000000000000008

L2ERC721Gateway: 0x5300000000000000000000000000000000000009

L2ERC1155Gateway: 0x530000000000000000000000000000000000000C

L2WETHGateway: 0x5300000000000000000000000000000000000010

L2ETHGateway: 0x5300000000000000000000000000000000000006

Other Layer 2 Predeploys

L2TxFeeVault: 0x530000000000000000000000000000000000000A

ProxyAdmin: 0x530000000000000000000000000000000000000B

MorphStandardERC20: 0x530000000000000000000000000000000000000D

MorphStandardERC20Factory: 0x530000000000000000000000000000000000000E

GasPriceOracle: 0x530000000000000000000000000000000000000F

L2WETH: 0x5300000000000000000000000000000000000011

2-how-to-run-a-morph-node.md:

---
title: How to Run a Morph Node
lang: en-US

Run a Morph Full Node

This guide outlines the steps to start a Morph node. The example assumes the home directory is /.morph

Hardware requirements

Running the morph node requires 2 processes:geth and node.

- Geth:the Morph execution layer which needs to meet the go-ethereum hardware requirements, but with less storage, 500GB is enough so far.

- Node:the Morph consensus layer embedded tendermint which needs to meet the tendermint hardware requirements.

:::tip
Due to limitations in the current geth implementation, only archive mode is supported, meaning the storage size will continually increase with produced blocks.
:::

Build executable binary

Clone morph

```
mkdir -p /.morph
cd /.morph
git clone https://github.com/morph-l2/morph.git
```

Currently, we use tag v0.2.0-beta as our beta version.

```
cd morph
git checkout v0.2.0-beta
```

Build Geth

Notice: You need C compiler to build geth

```
make ncccgeth
```

Build Node

```
cd /.morph/morph/node
make build
```

Sync from genesis block

Config Preparation

Download the config files and make data dir

```
cd /.morph
wget https://raw.githubusercontent.com/morph-l2/config-template/main/holesky/
data.zip
unzip data.zip
```

Create a shared secret with node

```
cd /.morph
openssl rand -hex 32 > jwt-secret.txt
```

Script to start the process

Geth

```
./morph/go-ethereum/build/bin/geth --morph-holesky \
    --datadir "./geth-data" \
    --http --http.api=web3,debug,eth,txpool,net,engine \
    --authrpc.addr localhost \
    --authrpc.vhosts="localhost" \
    --authrpc.port 8551 \
    --authrpc.jwtsecret=./jwt-secret.txt \
    --miner.gasprice="100000000" \
    --log.filename=./geth.log
```

tail -f geth.log to check if the Geth is running properly, or you can also

exeucte the below curl command to check if you are connected to the peer.

```
curl -X POST -H 'Content-Type: application/json' --data
'{"jsonrpc":"2.0","method":"netpeerCount","params":[],"id":74}'
localhost:8545
```


```
{"jsonrpc":"2.0","id":74,"result":"0x3"}
```


Node

```
./morph/node/build/bin/morphnode --home ./node-data \
     --l2.jwt-secret ./jwt-secret.txt \
     --l2.eth http://localhost:8545 \
     --l2.engine http://localhost:8551 \
     --log.filename ./node.log
```

tail -f node.log to check if the node is running properly, and you can also execute the command curl to check your node connection status.

```
curl http://localhost:26657/netinfo
```


```
{
  "jsonrpc": "2.0",
  "id": -1,
  "result": {
    "listening": true,
    "listeners": [
      "Listener(@)"
    ],
    "npeers": "3",
    "peers": [
      {
        "nodeinfo": {
          "protocolversion": {
            "p2p": "8",
            "block": "11",
            "app": "0"
          },
          "id": "0fb5ce425197a462a66de015ee5fbbf103835b8a",
          "listenaddr": "tcp://0.0.0.0:26656",
          "network": "chain-morph-holesky",
          "version": "0.37.0-alpha.1",
          "channels": "4020212223386061",
          "moniker": "morph-dataseed-node-1",
          "other": {
            "txindex": "on",
            "rpcaddress": "tcp://0.0.0.0:26657"
          }
        },
        "isoutbound": true,
```


Check sync status

curl http://localhost:26657/status to check the sync status of the node

```
{
  "jsonrpc": "2.0",
  "id": -1,
  "result": {
    "nodeinfo": {
      "protocolversion": {
        "p2p": "8",
        "block": "11",
        "app": "0"
      },
      "id": "b3f34dc2ce9c4fee5449426992941aee1e09670f",
      "listenaddr": "tcp://0.0.0.0:26656",
      "network": "chain-morph-holesky",
      "version": "0.37.0-alpha.1",
      "channels": "4020212223386061",
      "moniker": "my-morph-node",
      "other": {
        "txindex": "on",
        "rpcaddress": "tcp://0.0.0.0:26657"
      }
    },
    "syncinfo": {
      "latestblockhash":
"71024385DDBEB7B554DB11FD2AE097ECBD99B2AF826C11B2A74F7172F2DEE5D2",
      "latestapphash": "",
      "latestblockheight": "2992",
      "latestblocktime": "2024-04-25T13:48:27.647889852Z",
      "earliestblockhash":
"C7A73D3907C6CA34B9DFA043FC6D4529A8EAEC8F059E100055653E46E63F6F8E",
      "earliestapphash": "",
      "earliestblockheight": "1",
      "earliestblocktime": "2024-04-25T09:06:30Z",
      "catchingup": false
    },
    "validatorinfo": {
      "address": "5FB3D3734640792F14B70E7A53FBBD39DB9787A8",
      "pubkey": {
        "type": "tendermint/PubKeyEd25519",
        "value": "rzN67ZJWsaLSGGpNj7HOWs8nrL5kr1n+w0OckWUCetw="
      },
      "votingpower": "0"
    }
  }
}
```


The returned "catchingup" indicates whether the node is in sync or not. True means it is in sync. Meanwhile, the returned latestblockheight indicates the latest block height this node synced.

3-morph-json-rpc-api-methods.md:

---
title: Morph JSON-RPC API Methods
lang: en-US

Most methods are similar to Ethereum's. For those methods, we recommend you visit Ethereum JSON-RPC API.

This page lists some unique methods exclusive to Morph.

morphgetBlockByNumber

Returns information about a block by block number. In addition, it returns more fields than the standard ethgetBlockByNumber, such as withdrawTrieRoot, batchHash, nextL1MsgIndex and rowConsumption.

Parameters

1. QUANTITY|TAG - integer of a block number, or the string "earliest", "latest", "pending", "safe" or "finalized", as in the default block parameter.
2. Boolean - If true it returns the full transaction objects, if false only the hashes of the transactions.

Returns

See JSON-RPC API | ethereum.org, and more fields showing as the blew

- withdrawTrieRoot: DATA, 32 Bytes - the root of the withdraw trie, used to prove the users’ withdrawals.
- batchHash: DATA, 32 Bytes - the hash of the latest batch. It indicates the block is a batch point if it is not empty.
- nextL1MsgIndex: quantity - the next expected L1 message nonce after this block.
- rowConsumption: the rows consumption of this block, which rows are used to generate the ZK Proof based on halo2 schema.

Example

js
```
// request
curl -X POST --data '{"jsonrpc":"2.0","method":"morphgetBlockByNumber","params":
["0x1b4", true],"id":1}'

// Result
{
    "jsonrpc": "2.0",
    "id": 1,
    "result": {
        "batchHash":
"0x0000000000000000000000000000000000000000000000000000000000000000",
        "difficulty": "0x0",
        "extraData": "0x",
        "gasLimit": "0x989680",
        "gasUsed": "0x0",
        "hash":
"0xbc979055d001fe70ed637edd20e918bc583c84c35372f4cdf04253ec34b99178",
        "logsBloom":
"0x00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000",
        "miner": "0x0000000000000000000000000000000000000000",
        "mixHash":
"0x0000000000000000000000000000000000000000000000000000000000000000",
        "nextL1MsgIndex": "0x0",
        "nonce": "0x0000000000000000",
        "number": "0x1b4",
        "parentHash":
```

```
"0xff26c60bca2d08d9b0d17431a4c9d80d007dace61fb551bdf7c376d16bc77441",
        "receiptsRoot":
"0x56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421",
        "rowConsumption": [
            {
                "name": "evm",
                "rownumber": 2
            },
            {
                "name": "state",
                "rownumber": 4
            },
            {
                "name": "bytecode",
                "rownumber": 0
            },
            {
                "name": "copy",
                "rownumber": 4
            },
            {
                "name": "keccak",
                "rownumber": 1591
            },
            {
                "name": "tx",
                "rownumber": 0
            },
            {
                "name": "rlp",
                "rownumber": 0
            },
            {
                "name": "exp",
                "rownumber": 150
            },
            {
                "name": "modexp",
                "rownumber": 0
            },
            {
                "name": "pi",
                "rownumber": 0
            },
            {
                "name": "poseidon",
                "rownumber": 1222
            },
            {
                "name": "sig",
                "rownumber": 0
            },
            {
                "name": "ecc",
                "rownumber": 0
            },
            {
                "name": "mpt",
                "rownumber": 101
            }
        ],
        "sha3Uncles":
"0x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347",
        "size": "0x1fe",
```

```
        "stateRoot":
"0x1492cc1cebd586279388370e1184960e289d180eb867aa076fbad54aeb0a855b",
        "timestamp": "0x6619043f",
        "totalDifficulty": "0x0",
        "transactions": [],
        "transactionsRoot":
"0x56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421",
        "uncles": [],
        "withdrawTrieRoot":
"0x27ae5ba08d7291c96c8cbddcc148bf48a6d68c7974b94356f53754ef6171d757"
    }
}
```

morphgetBlockByHash

Returns information about a block by hash.

Parameters

- DATA, 32 Bytes - Hash of a block.
- Boolean - If true it returns the full transaction objects, if false only the hashes of the transactions.

Returns

See morphgetBlockByNumber returns

morphestimateL1DataFee

Generates and returns an estimate of how much L1DataFee the transaction will cost.

Parameters

1. Object - TransactionArgs
      - from: DATA, 20 Bytes - (optional) The address the transaction is sent from.
- to: DATA, 20 Bytes - The address the transaction is directed to.
- gas: QUANTITY - (optional) Integer of the gas provided for the transaction execution. ethcall consumes zero gas, but this parameter may be needed by some executions.
- gasPrice: QUANTITY - (optional) Integer of the gasPrice used for each paid gas.
- value: QUANTITY - (optional) Integer of the value sent with this transaction.
- input: DATA - (optional) Hash of the method signature and encoded parameters.

2. QUANTITY|TAG - integer block number, or the string "latest", "earliest", "pending", "safe" or "finalized".

Returns
QUANTITY - integer of the current l1 data fee in wei.

Example

```
js
// Request
curl -X POST --data
'{"jsonrpc":"2.0","method":"morphestimateL1DataFee","params":[{see
above}],"id":1}'

// Result
```

```
{
  "id":1,
  "jsonrpc": "2.0",
  "result": "0x3f4e2160f00"
}
```

morphgetNumSkippedTransactions

Get the number of the skipped transactions

Parameters

None

Returns

Quantity - integer of the number of the skipped transactions

Example

```
js
// Request
curl -X POST --data
'{"jsonrpc":"2.0","method":"morphgetNumSkippedTransactions","params":[],"id":1}'

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": "0"
}
```

morphgetSkippedTransactionHashes

Get a list of skipped transaction hashes between the two indices provided (inclusive)

Parameters

1. from index
2. to index

Returns

Arrays of transaction hashes

Example

```
js
// Request
curl -X POST --data
'{"jsonrpc":"2.0","method":"morphgetSkippedTransactionHashes","params":[0,
1],"id":1}'

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": [“0x….”,”0x….”]
}
```

morphgetSkippedTransaction

Get a skipped transaction by its hash.

Parameters

DATA, 32 Bytes - Hash of a transaction.

Returns

Object - a transaction object(see JSON-RPC API | ethereum.org), an some extended fields:
- sender: DATA, 20 Bytes - address of the sender.
- queueIndex: QUANTITY - integer of the L1Message index.
- skipReason: string - the skip reason of this transaction.
- skipBlockNumber: quantity - integer of the block number where the transaction is skipped.
- skipBlockHash: DATA, 32 Bytes - hash of the block where the transaction is skipped.

Example
js
// Request
curl -X POST --data
'{"jsonrpc":"2.0","method":"morphgetSkippedTransaction","params":
[“0x…”],"id":1}'

// Result

```
{
  "id":1,
  "jsonrpc": "2.0",
  "result": {
    "from": "0x59b79a4fc3e0e1646b254fb6804d8f5fa8fae7e0",
    "gas": "0x33450",
    "gasPrice": "0x0",
    "hash":
"0x74e13cc35fbeffc9747dd3a5a191b2d62939ee1d1c24651adb89b470f9cf0a36",
    "input":
"0x8ef1332e000000000000000000000082c74cd310438d4938799af4c2a29587284a2ca700000
0000000000000000005300000000000000000000000000000000000006000000000000000000000
0000000000000000000000000de0b6b3a764000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000a0000000000000000000000000000000000000000000000000000000000a4232e8
748000000000000000000000000f1d598fd5f8367be41b0761696e643ac092b313e0000000000000
00000000000f1d598fd5f8367be41b0761696e643ac092b313e000000000000000000000000000000
0000000000000000000de0b6b3a764000000000000000000000000000000000000000000000000000
0000000000000000800000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000",
    "nonce": "0x0",
    "to": "0x5300000000000000000000000000000000000007",
    "transactionIndex": "0x0",
    "value": "0x0",
    "type": "0x7e",
    "v": "0x0",
    "r": "0x0",
    "s": "0x0",
    "sender": "0x59b79a4fc3e0e1646b254fb6804d8f5fa8fae7e0",
    "queueIndex": "0x0",
    “skipReason”: “row consumption overflow”,
"skippedBlockHash":"0x61411498b0c241895d44c1bf9a9448ae8584bf434246d680980e8a5edf
```

```
9ce27c",
    "skippedBlockNumber": "0x382e",
    ""
}
}
```

morphgetSkippedTransactionHashes

Get the batches by index

Parameters

Quantity - integer of the index of batch

Returns

Object - RollupBatch
1. version: quantity - the version of the batch
2. hash: DATA: 32 bytes - the batch of this batch
3. parentBatchHeader: bytes - the parent batch header
4. Chunks: arrays of chunk - Chunk: bytes : the chunk bytes of this batch
5. skippedL1MessageBitmap: bytes - the bitmap of the skipped L1Message
6. prevStateRoot: DATA: 32 bytes - the state root at the beginning of this batch
7. postStateRoot: DATA: 32 bytes - the state root at the end of this batch
8. withdrawRoot: DATA : 32 bytes - the withdraw root at the end of this batch
9. sidecar - the side car for the rollup transaction of the blob type
10. signatures - array of object of signature:

- version: quantity - the version of sequencers
- signer: quantity - the index of the registered sequencer on L1Sequencer
- signerPubKey: bytes - the bls public key
- signature: the bls signature of the batch signed by this signer

Example
```
js
// Request
curl -X POST --data
'{"jsonrpc":"2.0","method":"morphgetRollupBatchByIndex","params":[1],"id":1}'

//Result
{
    "jsonrpc": "2.0",
    "id": 1,
    "result": {
        "version": 0,
        "hash":
"0x559c75fb09451ef19968b6c17c2ed3b3f431ab496d29d0a611970bf3e377ed4f",
        "parentBatchHeader":
"0x000000000000000000000000000000000000000000000000008b1cfd86e7ac16004ef7aae3e69
3402764e10471efc9a67c3051a44b2d5a5ad2010657f37554c781402a22917dee2f75def7ab966d7
b770905398eba3c444014000000000000000000000000000000000000000000000000000000000
00000",
        "chunks": [

"0x0100000000000000001000000006623829f0000000000000000000000000000000000000000
0000000000000000000000000000098968000000000"
        ],
        "skippedL1MessageBitmap": "0x",
        "prevStateRoot":
"0x232ce5fc696e7033ec630174d4c78aa157cc13ffba01aba5154bb900ae2b61b3",
        "postStateRoot":
"0x232ce5fc696e7033ec630174d4c78aa157cc13ffba01aba5154bb900ae2b61b3",
```

```
        "withdrawRoot":
"0x27ae5ba08d7291c96c8cbddcc148bf48a6d68c7974b94356f53754ef6171d757",
        "sidecar": {
            "Blobs": [
                [...]
            ],
            "Commitments": [
                [...]
            ],
            "Proofs": [
                [...]
            ]
        },
        "signatures": [
            {
                "version": 0,
                "signer": 0,
                "signerPubKey":
"0x000000000000000000000000000000000095ad465c2895ee825c7d4f1b60a18734db57d4108369
e47c6e3a94ee15846f825c06dad5d98f503bd31ece1d9f94b1100000000000000000000000000000000
0000c5d6ba04bc9b9674dd2acbfc5caed3976c1b8be2ec90a03d78dffe924648b4fba82225aff43c
744310c6a60185b75ac0000000000000000000000000000000fce6be001c871a11b9db1c6c15f0
a6999de5646941a74486206dc784f0b3ffe11799212f3f44ef754b4a0f1ecf856390000000000000000
0000000000000000000b2f06634e5ea719682c30911c94dfb560f0b7656b5c34a871ea035e3fe7b
041885420f8fe1e251f1cce5cdb7514869e",
                "signature":
"0x079ff817623be61e3dc48ad755020d3eb37e47697e97394df3707248e5e923d8cb92cc2c788e2
e19eb21d878224e3e2a023170f091b979dbdc5ac47e95a1e3e6edf5a91279b3c89e871927370abee
fd43430b561b640d91f0f474db8e871f05f"
            },
            {
                "version": 0,
                "signer": 1,
                "signerPubKey":
"0x000000000000000000000000000000010173aeac4ff317e8e60493f962b91dbd27614e1f6594
e17d18a02968bd1fd698b6703092ab8622cd22d6948d9421156000000000000000000000000000000
0000801aea15697ab4d7a808be45377e4f0d2f54857fdc04031e476402ff16c66a6cbcc5f09a84bf
85400c8afbabed00660000000000000000000000000000000015fc71b2c4e81148274e6169c9c9a
ace8c34fa6030547650242b6c32527dd23a996416e32640bce4f495a0afabc7dbb900000000000000
0000000000000000000088c4a0dffccc96bce47aef0e176b129457a5f3ae1651b132ddb418e9f7b5
850a38c6fec1be6d169eb88dc1619648bf4",
                "signature":
"0x05784fcf2c9b50b2503282a464106ce6bad4d29798869d2d24dae047eba48662ef66b54f193a4
a0aad511cee57b8fc960b44e478cfc0c021a92dba44495ffa9237f4cf4f9865bb8fb95cee9bb32f2
b283940bce6d8f4fdfe89bec0c186ecc429"
            },
            {
                "version": 0,
                "signer": 2,
                "signerPubKey":
"0x000000000000000000000000000000003fd9468a8ceffc1b696874517777ef8bfdc9a1bade95
c480ee2624903e648c1caf01c65de5b4fda8876a3a0e8d9f0890000000000000000000000000000000
00004c02f3609a0f61d12fe737dcbb047d5253bd3ff905b55c0e0f932b476fd77d172a58b72ef0f5
06407870988dd6038220000000000000000000000000000000017fa5765899f60f7a58f8ccdaaa29
5cde55992231710672692ba6a71a4faa9572f728f438ded65576a570d57e19fd3040000000000000
0000000000000000000001226138813bde98af3464ed03649d8c731bc4e5cb3d26b53bf7483f4105d1
8bbb3f19e23905119e156e7d003d2fd125c",
                "signature":
"0x17d127f711bbfd578f3a9d6c897f6eb4c109f2b12ec16c7976c2146325b718ac06713ecfd168b
1abd60d5c4d84b8c75105019a522a96f9d28974dafab640e146abc29322e5550bdb606bcf768c1d7
de468fa42e09f2c806bfbedcffd6cfa652c"
            },
            {
                "version": 0,
```

```
                "signer": 3,
                "signerPubKey":
"0x000000000000000000000000000000000109bf02a2636c0dc1968b0a50db77251eb090c3e9f51e
2a2bc60c4ac72213f41f01f0a34e92c2e0625bd62e28e27edb500000000000000000000000000000000
000139969bd92522113c0615659874d1fae311ad8152d0584c7b57ffc14927067486dcf86413c568
4fccc1163ee2d45c1c1000000000000000000000000000000000f172603f70a0730d100ad6d28bde
477195987062e8ade83b82d093935d956ff20ca768c26263577b094f1cb756adc400000000000000
0000000000000000000010dde3acca00b4ff1b4976500a8f97e92246f43f78cadc95c4993dfc4f4c5
01c33d42a4bf52587f4931287b59623149c",
                "signature":
"0x053a7119c0fa920e94c8a1f32441537509f654a673af02179a01ff2c7ac6dee24b22c2f93da4b
4a37bb08e2115ef097e00038d0cabdeec2ae1838aa30cb62bff8e528ac200eff0d69fbe884864697
2dbb09f4cf3a06c62be0c5e314f25bdbd68"
            }
        ]
    }
}
```


morphGetBlockTraceByNumberOrHash

It replays the block and returns the structured blockTrace for rollers.

Parameters

1. QUANTITY|TAG - integer of a block number, or the string "earliest", "latest", "pending", "safe" or "finalized", as in the default block parameter. Or DATA, 32 Bytes - Hash of a block.
2. tracerConfig(optional), see debug Namespace | go-ethereum

Example

js

```
// Request
{"jsonrpc":"2.0","method":"morphgetBlockTraceByNumberOrHash","params":
["latest"],"id":67}

// Result
{
    "jsonrpc": "2.0",
    "id": 67,
    "result": {
        "chainID": 53077,
        "version": "5.2.0-mainnet",
        "coinbase": {
            "address": "0xfabb0ac9d68b0b445fb7357272ff202c5651694a",
            "nonce": 0,
            "balance":
"0x20000000000000000000000000000000000000000000000006b49161ba10",
            "keccakCodeHash":
"0xc5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470",
            "poseidonCodeHash":
"0x2098f5fb9e239eab3ceac3f27b81e481dc3124d55ffed523a839ee8446b64864",
            "codeSize": 0
        },
        "header": {
            "parentHash":
"0xa2b3ee7a3718baeb1b460bc9a479838532c184129d0238b342e1bc9430e15961",
            "sha3Uncles":
"0x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347",
            "miner": "0x0000000000000000000000000000000000000000",
            "stateRoot":
"0x1492cc1cebd586279388370e1184960e289d180eb867aa076fbad54aeb0a855b",
```

            "transactionsRoot":
"0x56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421",
            "receiptsRoot":
"0x56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421",
            "logsBloom":
"0x00000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000",
            "difficulty": "0x0",
            "number": "0x31",
            "gasLimit": "0x989680",
            "gasUsed": "0x0",
            "timestamp": "0x6618fa6a",
            "extraData": "0x",
            "mixHash":
"0x0000000000000000000000000000000000000000000000000000000000000000",
            "nonce": "0x0000000000000000",
            "nextL1MsgIndex": "0x0",
            "batchHash":
"0x0000000000000000000000000000000000000000000000000000000000000000",
            "baseFeePerGas": null,
            "withdrawalsRoot": null,
            "blobGasUsed": null,
            "excessBlobGas": null,
            "parentBeaconBlockRoot": null,
            "hash":
"0x65645773fa2056771753878cdf5a1280dff36756096e8f9c1cf51fe66690a448"
        },
        "transactions": [],
        "storageTrace": {
            "rootBefore":
"0x1492cc1cebd586279388370e1184960e289d180eb867aa076fbad54aeb0a855b",
            "rootAfter":
"0x1492cc1cebd586279388370e1184960e289d180eb867aa076fbad54aeb0a855b",
            "proofs": {
                "0x5300000000000000000000000000000000000001": [

"0x09261068d5568cdf6020d6b6703b831fbc4c5928019dcc23606eceaa4cb4befd3a22bd9a05e5f
4619cebb743688e2bb2510352fa7b64cee7d6c947e1c8655b3e88",

"0x0912cea39ac19eee30407d6760f267dcaa3456c6ab30a2d7285e1ae726debdc3a228bc075d0a3
a84f79e301293ac8c4188d5e58e3a30a49ca058d77784ec477d6c",

"0x092a1073697a898cbfc740c3c327cc3bcc1517b23dd5a44ea7f7e5975a55a615790e69dfcd0c8
346b28ff2dffa8db5d0c5b7ed8291eab83be0292051c2dd9a55bc",

"0x0927f8c131313ff2ce0d551273c407fd6e94e5d9f644eab09c4c20b13e00fb5a4b1e00b9701c7
f2ac25be7e3fe8ad00ce18f642d07188ff1aacc6d7a1fc44987b5",

"0x090296736b61c7a9c43b7260c2f652d1bc941b882c0a1e71a9b8d86657cc0b9d8e1b6b72e48fe
7cfc89ae526ae345245d7f6871db82148a8a0c1c9cf992e177ba2",

"0x090a6feef711af23824583c2bcd54147c2991287ca1fbad2cbb0e9ac2f42dc0c2d1b510d4e260
02d7f33e82092a7c28f26ac6cf3f2349eb23e4374ad31f3e289d0",

"0x091a8fc20731feea420de908b0d66e7c53398f53ce5d76054cf3d0e4747ba1896222d288ea2a9
d771a19771658abf5a64fd503f4ed277aa8d6f779137a88ea1c8d",

"0x091628b91a861616a64807781ae6aaa549dae9555ef1454a85df052398404f44690f386bb59d1
06d7a27ff2df1de738c3fae5cf8307f84e07ac6957e3c42fa3f5e",

```
"0x09220b163279be211a426832f5ff47e0ecadb5bd5eaaf0d62189998fde4e14e62913696e7ffd5
60b4bf7df760cbfdc330a8c5e7cf15b4e2f60fb9146fd28fe2db7",

"0x08150feb81fe1639286c75adb93e5467d36ad52d9ff00e0c706bc5345f6dcd495000000000000
0000000000000000000000000000000000000000000000000000000000000000",

"0x081cb430cbc5dd11f62a713fb3656184085386104dc53fc135fb3c56dba496705000000000000
0000000000000000000000000000000000000000000000000000000000000000",

"0x060c320ff8f6c3907ae6df62a7fe70e2be093390bd246197d5f56b6bce56f513f812fac8f51e3
812914c097147217411381b9b2dad76963ef63c261885d3a49223",

"0x0427cfcbd2fe87474008e199ef5bff09273803bbf38032c75a36a7726b3c88f8a205080000000
0000000000000000000000000000000000000000aec0000000000000000000000000000000000000
0000000000000000000000000000000000000000000002198e2c16011cd18ef5be82557ea4552a16
cc88ed8026d68fc38a18776550fa944107d7c70e67e04d35d77455d5755c94d4897196e4ea9a341c
1a6d864a6d4d518d32c9790b7656cb1991e831f76e6d65151cbd1c05aef8cfa7d369ad2076f8400"
,

"0x5448495320495320534f4d45204d4147494320425954455320464f5220534d54206d317252586
7503278704449"
                ],
                "0x530000000000000000000000000000000000000f": [

"0x09261068d5568cdf6020d6b6703b831fbc4c5928019dcc23606eceaa4cb4befd3a22bd9a05e5f
4619cebb743688e2bb2510352fa7b64cee7d6c947e1c8655b3e88",

"0x0912cea39ac19eee30407d6760f267dcaa3456c6ab30a2d7285e1ae726debdc3a228bc075d0a3
a84f79e301293ac8c4188d5e58e3a30a49ca058d77784ec477d6c",

"0x0910b680ab891bca41519a50437acf0fd6001d3843132362201baeecfefe64861d004b217e8ef
d3e9ff7f15214f53868836cd5341e524a6862cf599a2c91e066e4",

"0x092bc917a3803350b7b4b99e33f5612886fe67ae2346b7b8a80c4f33ea2b2143c5157889ebd8b
d189e158f8e315bf0e81eeaade2de06d9514962a676c2b5c43fe6",

"0x0923373510953f88897d7fcfd22182f41afbc558d2f862f437a31d1919ce2e34c925c888335ad
923fc91289b73b7cb65de69d93146389a6af856490d8578203517",

"0x092ea32b06c112526e73d5c4828866500f096d35e79c13d7c1f7aa42afce901bd618bb82398f0
bc28770139185439bf42dea1f543e8a0a266399ba43a3dd60f463",

"0x09199e7116f18fc4121b627c11b0ca83bbb70b204098c01db84cd2c27ec3416bd4002ea098b20
a57201744454a787c20550c1ffa4145ab218c28906ba3df5ea6cb",

"0x0923bfab6f7b9cc20bae8ca8989ad45df9625cf0c14ee54257ffd31b2593b8385f0861f83d52b
30f3b4ea0dbf9aa1b95e8864f2b2032249d6fa43ba22be66e5e82",

"0x0609fd578e322ce608e81c29d820eca5e2d610bf211aa40fbfdfc7bd1327c9c6432a868ae99ff
40d43cff3f28ff0655bf467eaa89e3d3a1493203e4abb26099b3b",

"0x0418a86f98026d1a55f6fee616f04d51be16c033651ffaafb0514287ec067d04e805080000000
0000000000000000000000000000000000000000aec0000000000000000000000000000000000000
0000000000000000000000000000000000000000000025f5484cf6758b2d9e10b66230e7ed93fd
790931c579d1583b40a25c29b3cc944107d7c70e67e04d35d77455d5755c94d4897196e4ea9a341c
1a6d864a6d4d518d32c9790b7656cb1991e831f76e6d65151cbd1c05aef8cfa7d369ad2076f84205
3000000000000000000000000000000000000f0000000000000000000000",

"0x5448495320495320534f4d45204d4147494320425954455320464f5220534d54206d317252586
7503278704449"
                ],
                "0xFABB0ac9d68B0B445fB7357272Ff202C5651694a": [
```

```
"0x09261068d5568cdf6020d6b6703b831fbc4c5928019dcc23606eceaa4cb4befd3a22bd9a05e5f
4619cebb743688e2bb2510352fa7b64cee7d6c947e1c8655b3e88",

"0x09132e097dde889d85b8b3309a827f0496c58979f869fe9827851d59f3af8d3bad1db74244979
778c4e7c6b56496b5a3f1ee84cd42acacbc6ae3842deb5332b101",

"0x0919474ec6b34f96f735c336cc161ed84fdd3a683fa201b518af5c2c1626f02f4612a08d77db2
b9b67c38d89ed19f9ca59dadbd1699d27829f985020f6f3432959",

"0x090dc9f8dd06d98c5a78af9836fc75823e4e2b0c95085c36dc33b8c35f5fc224d319e43044e7a
c03d516f20d38f68682109407871d5f07f080c59857789fad1e46",

"0x0907bc083c80445c99e7239f960beabcf360f7a6999ade44323b8401e0ec6842b80c789c2f71e
21b35f96a71c66a774edac37ba18b809b8122d9000fbf0f8ac74b",

"0x091e9ddabce6463c4843a7f43ce562b40bd7cc778a539486f6739bf7daf6f14bdf2da1168d26e
898c81e03ba275f3474d4172c1f936d272114b87a486ae25a0c7e",

"0x092ebed3e1476b235ef7ee7d5c160d7738aee3a9ecf02754975430203dbe05b7ce1f81e74d1f8
a355b7dda2b293a8833c0926df34b6ab6516de484a7c682123e3b",

"0x0700000000000000000000000000000000000000000000000000000000000000bca6f26c45
2ec72dc90f8ac79855df88d3b88ce77a5c4ac472eea9ef7b619ae",

"0x0824eb7acc6f05bd88d0aecfb7e170edc0f99384a54657916103d91fad99bf010300000000000
0000000000000000000000000000000000000000000000000000",

"0x072449731363c43de43d2516f39791c43a3be1e83edc3432352d81d2d1384a946e159ad10b4c8
9ff16157cc364c81f6039eeea9a8eb87e4b4700249219e71cc3ef",

"0x0403d2c88868dc8b5f7994604147a8246c48cc905235de980c42c3e78f351f00c505080000000
0000000000000000000000000000000000000000000000000000000020000000000000000
0000000000000000000000000000006b49161ba1000000000000000000000000000000000
0000000000000000000000000c5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bf
ad8045d85a4702098f5fb9e239eab3ceac3f27b81e481dc3124d55ffed523a839ee8446b6486420f
abb0ac9d68b0b445fb7357272ff202c5651694a0000000000000000000000000",

"0x5448495320495320534f4d45204d41474943204259544553204 64f5220534d54206d317252586
7503278704449"
                ]
            },
            "storageProofs": {
                "0x530000000000000000000000000000000000000000001": {

"0x0000000000000000000000000000000000000000000000000000000000000021": [

"0x0825bcd8ed4a7e39c0f0e835761cda36131dd34f16de95765d2216d997923a52f60e38bf5d7d5
92ab89df92d87f260b8529d9c466b936b42ad8fbfe576bf97b200",

"0x06216372aa3165a29ee67494a5a9680358be92c254cc5724949e4d52742ba8d57f02a40b56146
37fca2783d673559f035f5f5a0588ee27d164bb5375568493200a",

"0x042f683d080c393f5fb2ade2190fd56e8a5e8fa26533acdea8cce25caa236027cc0101000027a
e5ba08d7291c96c8cbddcc148bf48a6d68c7974b94356f53754ef6171d75700",

"0x5448495320495320534f4d45204d41474943204259544553204 64f5220534d54206d317252586
7503278704449"
                    ]
                },
                "0x530000000000000000000000000000000000000000f": {

"0x0000000000000000000000000000000000000000000000000000000000000001": [

"0x09224baa919e1495600105d8a4446f80cabd9a915d7556e0b5d2956a3ad01e96862b531affe57
```

```
e7f49ddcfef93ff839a215de0865c985188f99c61838f366e2bbe",

"0x091eb325ae1568060f3957cc990d35e5ef48b7b2bc7979151083abe2d68aabc812162b4d479ff
13f2af3f0cfaadc6bccc8622dd980f83c7014e947f50577eedaba",

"0x08015c4fc60a3c6160762783cfc0da7e9d0518e56f0f4f236b7666e1afde43c8f60e867960e70
3abfccf265c16aceb5c2a092c39248e00fa04f717cf3ca38dfe9e",

"0x080782addd8b48a93860ab4cb45e80c6cc869c1288f09d52cb7dec1f6aab62deb700000000000
0000000000000000000000000000000000000000000000000000",

"0x0700000000000000000000000000000000000000000000000000000000000000000000f35c10e189
3b6065c8ce8ba23fcd841bb168eb35643ca124118a4e44668fab2",

"0x080487bd50c628c019d7858418000d1393427df64c9c103a4229f7b6cb3cdf5ae600000000000
0000000000000000000000000000000000000000000000000000",

"0x0700000000000000000000000000000000000000000000000000000000000001046b31bcd3
1548fa80d0a9fb9fae45e61112d689f6b9c82958033c0787f1f94",

"0x0805b5be5c8d36bd01ef9c3d10f3d994da611bc1484e9e48f69d2224efeb811b3200000000000
0000000000000000000000000000000000000000000000000000",

"0x081ebc02f6986209f10fc7d04ba0c4ef196d75ad0a22e786fc7cf789f591e9b26f00000000000
0000000000000000000000000000000000000000000000000000",

"0x0700000000000000000000000000000000000000000000000000000000002680dd0a6ed
49fc1faf7d4ef23a8328de46b8a4c1d86e6553f4940e394673978",

"0x0628658ea3cbfa933d0e4041f9000bcc3393a3536913008ef6e1837ba7053bdec1156a0960a84
fa4287f66d18b9c39ab9d403fa6b6dddd126f47758aadcb82e122",

"0x0426049ba6de63003492eb078a01a8aa4f4a0e67f28f0955c2eba9101d5d2eea5001010000000
0000000000000000000000000000000000000000000000000700",

"0x5448495320495320534f4d45204d41474943204259544553204 64f5220534d54206d317252586
7503278704449"
                    ],

"0x0000000000000000000000000000000000000000000000000000000000000002": [

"0x09224baa919e1495600105d8a4446f80cabd9a915d7556e0b5d2956a3ad01e96862b531affe57
e7f49ddcfef93ff839a215de0865c985188f99c61838f366e2bbe",

"0x091eb325ae1568060f3957cc990d35e5ef48b7b2bc7979151083abe2d68aabc812162b4d479ff
13f2af3f0cfaadc6bccc8622dd980f83c7014e947f50577eedaba",

"0x08015c4fc60a3c6160762783cfc0da7e9d0518e56f0f4f236b7666e1afde43c8f60e867960e70
3abfccf265c16aceb5c2a092c39248e00fa04f717cf3ca38dfe9e",

"0x04020953ad52de135367a1ba2629636216ed5174cce5629d11b5d97fe733f07dcc01010000000
0000000000000000000000000000000000000000000000029d72000000000000000000
0000000000000000000000000000000000000000000000000002",

"0x5448495320495320534f4d45204d41474943204259544553204 64f5220534d54206d317252586
7503278704449"
                    ],

"0x0000000000000000000000000000000000000000000000000000000000000003": [

"0x09224baa919e1495600105d8a4446f80cabd9a915d7556e0b5d2956a3ad01e96862b531affe57
e7f49ddcfef93ff839a215de0865c985188f99c61838f366e2bbe",

"0x0700000000000000000000000000000000000000000000000000000000014a36870c64
```

d2edad5a210b41e5bdfd73e7f4201161a755c434a621ffe103cc8",

"0x061c3181dc54d9c120fdc034e9c12d47d19f07bdeb049e492e0e7c5c9d7fe19a4d0e38bf5d7d5
92ab89df92d87f260b8529d9c466b936b42ad8fbfe576bf97b200",

"0x0406c50541f08911ad149aa545dd3d606f86ee63c751a795c7d57f0d3f85e6bdeb01010000000
0000000000000000000000000000000000000000000000174876e8002000000000000000000
0000000000000000000000000000000000000000000003",

"0x5448495320495320534f4d45204d41474943204259544553204 64f5220534d54206d317252586
7503278704449"
                    ]
                }
            }
        },
        "executionResults": [],
        "withdrawtrieroot":
"0x27ae5ba08d7291c96c8cbddcc148bf48a6d68c7974b94356f53754ef6171d757",
        "startL1QueueIndex": 0
    }
}

4-create-a-safe-multi-signature-wallet.md:

---
title: Create a Safe Multi-Signature Wallet
lang: en-US

What is Morph Safe?

!safemultisig

Morph Safe is a fork managed by the Morph team in collaboration with Protofire, an official partner of the Safe team. It is a multi-signature smart contract wallet, which means it requires a minimum number of signatures to approve a transaction before it can occur.

For example, if your business has three main stakeholders, you can configure the wallet to require approval from either two out of three or all three individuals before a transaction can proceed. This security measure ensures that no single person can compromise the funds.

In addition to robust security, Morph Safe offers complete self-custody over funds. The smart contract deployed for the wallet has full control over the wallet's contents and does not rely on any trusted external party for execution.

To create your Morph Safe wallet click here.

6-ecosystem-infrastructure.md:

---
title: Use Ecosystem Infrastructures
lang: en-US

Besides the official tools and resources, we have also partnered with other infrastructure projects to help builders on Morph. Here is the list of the projects that are live on Morph Holesky testnet.

:::tip More partners coming soon

As Morph is still in beta testnet stage, we are working closely to complete the infrastructure cooperations, to ensure clear instructions, patners that is still working on Holesky testnet integration is not listed, we will keep updating this page.
:::

<!--
3rd Party Bridges

[LayerZero]()

[Orbiter Finance]()

[Axelar]()
-->

RPC Services

Quicknode

QuickNode is transforming blockchain infrastructure and tooling by simplifying web3 development and providing high-performance access to Morph.

Developers in the Morph ecosystem are now eligible for free QuickNode credits/discounts!

Read more about what you can get here:

Quicknode Partnership

!QuickNode

Account Abstraction

Biconomy

!biconomy

Documentation: https://docs.biconomy.io

Integration for Morph Holesky is still in process, more details coming soon.

Blockchain Indexing Services

Goldsky

!goldsky

We have partnered with Goldsky to provide indexing and subgraph services for Morph Holesky and Morph Mainnet.

Goldsky also allow us to grant selected developers with free subgraph services now. If you want to deploy a subgraph with Goldsky, please raise a ticket in our Discord to let us know how we can help.

For more information, please refer to their documentation.

Oracles

Pyth Network

Documentation: https://docs.pyth.network/price-feeds

Right now Pyth is deployed on Morph Holesky, check the contract.

eOracle

Deployed contract

Full price feeds and docs can be found here : https://eoracle.gitbook.io/eoracle/price-feeds/feed-addresses.

MultiSig

Morph Safe

We have partnered with Protofire to create Morph Safe for multisig services on Morph.

Please refer to the docs about How to create a safe multisig on Morph.

Explorer

Besides the official Morph explorer, we have also partnered with 3rd party providers for different Morph blockchain data presentations forms.

Socialscan

Track Morph Testnet transactions and wallets on the SocialScan Explorer.

!socialscan

7-dapp-examples-on-morph.md:

---
title: Dapp Examples on Morph
lang: en-US
keywords: [morph,ethereum,rollup,layer2,validity proof,optimistic zk-rollup]
description: Upgrade your blockchain experience with Morph - the secure decentralized, cost0efficient, and high-performing optimistic zk-rollup solution. Try it now!

We provide multiple example Dapps for developers, built on our Morph starter template, deployed on the Morph Testnet, and accompanied by corresponding frontends.

These examples are very helpful for developers to go through the entire development process on Morph. Below is a brief introduction of the example Dapps and tutorial links.

:::tip Morph Starter Template

The Morph starter kit helps developers quickly and efficiently, build dApps on the Morph blockchain. It is a comprehensive template for building fullstack dApps.

:::

Example Dapp: Building a Decentralized Hotel Booking System on Morph

!Okido Finance

Check the tutorial

This guide walks you through the process of building a decentralized hotel booking system on the MorphL2 blockchain using Solidity smart contracts and a React & Wagmi front-end. This system allows hotel owners to add rooms, manage room availability, accept bookings, and receive reviews from guests.

Example Dapp: Building a fractionalized real-estate dApp

!Okido Finance

Check the tutorial

Imagine being able to invest in a high-value real estate property without needing to shell out a huge amount of money upfront. Sounds pretty appealing, right? This is the magic of fractionalization. By dividing a property into smaller, more affordable shares, fractionalization makes it possible for just about anyone to get in on the real estate action. It’s like turning a luxury mansion into a bunch of reasonably priced slices that you can buy and sell easily, just like stocks.

Our project today, Okido Finance, takes this idea and builds a decentralized application (DApp) around it. With Okido Finance, property owners can create shares of their real estate assets and investors can buy these shares using a custom ERC20 token. This not only democratizes real estate investment but also adds a layer of liquidity, making it easier to trade shares and get in or out of investments.

In this tutorial, I’ll walk you through building the Okido Finance DApp step by step. We’ll start with setting up the development environment, move on to deploying smart contracts, and finish with designing a user-friendly UI.

By the end, you’ll have a solid grasp of how to build a decentralized real estate fractionalization platform. Whether you’re looking to create something similar or just want to learn more about these technologies, you’ll be well-equipped to dive deeper into fractionalization. Let’s get started!

Query and Index Smart Contracts on Morph using Goldsky

!Okido Finance

Check the tutorial

Picture this: you're a developer with a brand-new smart contract deployed on the Morph. Your next challenge is to efficiently query and retrieve on-chain data for your decentralized application (dApp). That's where subgraphs come in. Subgraphs offer a powerful and flexible way to index and query blockchain data, making it easy to build responsive and data-rich dApps.

In this tutorial, we'll walk through deploying a smart contract on the Morph chain and setting up a subgraph to query this contract. We’ll also introduce Goldsky, a tool that makes creating and managing subgraphs a breeze. Let’s get started and see what we can build together!

1-how-to-run-a-morph-node.md:

---
title: Run a Morph Full Node from Source

lang: en-US

This guide outlines the steps to start a Morph node. The example assumes the home directory is /.morph

Hardware requirements

Running the morph node requires 2 processes: geth and node.

- Geth:the Morph execution layer which needs to meet the go-ethereum hardware requirements, but with less storage, 500GB is enough so far.

- Node:the Morph consensus layer embedded tendermint which needs to meet the tendermint hardware requirements.

:::tip
According to limitations of the current geth implementation, we only support archive mode for launching a Geth. So the storage size of Geth will constantly increase along with blocks produced.
:::

Build executable binary

Clone morph

```
mkdir -p /.morph
cd /.morph
git clone https://github.com/morph-l2/morph.git
```

Currently, we use tag v0.2.0-beta as our beta version.

```
cd morph
git checkout v0.2.0-beta
```

Build Geth

Notice: You need C compiler to build geth

```
make ncccgeth
```

Build Node

```
cd /.morph/morph/node
make build
```

Config Preparation

1. Download the config files and make data dir

```
cd /.morph
wget https://raw.githubusercontent.com/morph-l2/config-template/main/holesky/data.zip
unzip data.zip
```

2. Create a shared secret with node

```
cd /.morph
openssl rand -hex 32 > jwt-secret.txt
```

Sync from snapshot(Recommended)

You should build the binary and prepare the config files in the above steps first, then download the snapshot.

Download snapshot

```
bash
download package
wget -q --show-progress https://snapshot.morphl2.io/holesky/snapshot-20240805-1.tar.gz
uncompress package
tar -xzvf snapshot-20240805-1.tar.gz
```

Extracting snapshot data to the data directory your node points to

```
bash
mv snapshot-20240805-1/geth geth-data
mv snapshot-20240805-1/data node-data
```

Start execution client

```
bash
./morph/go-ethereum/build/bin/geth --morph-holesky \
    --datadir "./geth-data" \
    --http --http.api=web3,debug,eth,txpool,net,engine \
    --authrpc.addr localhost \
    --authrpc.vhosts="localhost" \
    --authrpc.port 8551 \
    --authrpc.jwtsecret=./jwt-secret.txt \
    --log.filename=./geth.log
```

tail -f geth.log to check if the Geth is running properly, or you can also execute the curl command below to check if you are connected to the peer.

```
Shell
curl -X POST -H 'Content-Type: application/json' --data '{"jsonrpc":"2.0","method":"netpeerCount","params":[],"id":74}' localhost:8545

{"jsonrpc":"2.0","id":74,"result":"0x3"}
```

Start consensus client

```
Bash
 ./morph/node/build/bin/morphnode --home ./node-data \
     --l2.jwt-secret ./jwt-secret.txt \
     --l2.eth http://localhost:8545 \
     --l2.engine http://localhost:8551 \
     --log.filename ./node.log
```

tail -f node.log to check if the node is running properly, and you can also

execute the command curl to check your node connection status.

Bash
curl http://localhost:26657/netinfo


```
{
  "jsonrpc": "2.0",
  "id": -1,
  "result": {
    "listening": true,
    "listeners": [
      "Listener(@)"
    ],
    "npeers": "3",
    "peers": [
      {
        "nodeinfo": {
          "protocolversion": {
            "p2p": "8",
            "block": "11",
            "app": "0"
          },
          "id": "0fb5ce425197a462a66de015ee5fbbf103835b8a",
          "listenaddr": "tcp://0.0.0.0:26656",
          "network": "chain-morph-holesky",
          "version": "0.37.0-alpha.1",
          "channels": "4020212223386061",
          "moniker": "morph-dataseed-node-1",
          "other": {
            "txindex": "on",
            "rpcaddress": "tcp://0.0.0.0:26657"
          }
        },
        "isoutbound": true,
 .......
```


Check sync status

curl http://localhost:26657/status to check the sync status of the node

Bash

```
{
  "jsonrpc": "2.0",
  "id": -1,
  "result": {
    "nodeinfo": {
      "protocolversion": {
        "p2p": "8",
        "block": "11",
        "app": "0"
      },
      "id": "b3f34dc2ce9c4fee5449426992941aee1e09670f",
      "listenaddr": "tcp://0.0.0.0:26656",
      "network": "chain-morph-holesky",
      "version": "0.37.0-alpha.1",
      "channels": "4020212223386061",
      "moniker": "my-morph-node",
      "other": {
        "txindex": "on",
        "rpcaddress": "tcp://0.0.0.0:26657"
      }
    },
    "syncinfo": {
```

```
      "latestblockhash":
"71024385DDBEB7B554DB11FD2AE097ECBD99B2AF826C11B2A74F7172F2DEE5D2",
      "latestapphash": "",
      "latestblockheight": "2992",
      "latestblocktime": "2024-04-25T13:48:27.647889852Z",
      "earliestblockhash":
"C7A73D3907C6CA34B9DFA043FC6D4529A8EAEC8F059E100055653E46E63F6F8E",
      "earliestapphash": "",
      "earliestblockheight": "1",
      "earliestblocktime": "2024-04-25T09:06:30Z",
      "catchingup": false
    },
    "validatorinfo": {
      "address": "5FB3D3734640792F14B70E7A53FBBD39DB9787A8",
      "pubkey": {
        "type": "tendermint/PubKeyEd25519",
        "value": "rzN67ZJWsaLSGGpNj7HOWs8nrL5kr1n+w0OckWUCetw="
      },
      "votingpower": "0"
    }
  }
}
```


The returned "catchingup" indicates whether the node is in sync or not. True means it is in sync. Meanwhile, the returned latestblockheight indicates the latest block height this node synced.

Sync from genesis block(Not Recommended)
Start the execution client and consensus client directly without downloading snapshot

2-how-to-run-a-morph-node-docker.md:

---
title: Run a Morph Full Node with Docker
lang: en-US

This guide will help you start a full node running in the docker container.

Quick Start

Currently, users need to build the Docker image themselves using the Docker file and Docker Compose file we provide. However, there's no need to worry, as you only need one command to quickly start a full node. This command will handle everything for you, including downloading snapshots, structure data and config files, building the image, and starting the container.

1. Clone the dockerfile repository

bash
git clone --branch release/v0.2.x https://github.com/morph-l2/morph.git

2. Run the following command

bash
cd ops/publicnode
make run-holesky-node

Running this command will create a .morph-holesky directory in your user directory by default, serving as the node's home directory. Before starting the node, this command will perform several preparations:

- Create the node's home directory and copy the default configuration files into it.
- Prepare the secret-jwt.txt file for for authentication during RPC calls between geth and the node.
- Download the latest snapshot data to speed up node synchronization.
- Place the extracted snapshot data into the corresponding folder within the home directory.

After completing these preparations, the command will automatically build the image and start the container, with Docker volumes mounted to the created node home directory.

:::info
If this is your first run, these processes may take some time. Note that if you are starting the node for the first time but already have a .morph-holesky directory, you must delete that directory before running the command. Otherwise, the preparation phase will be skipped, which may prevent the node from running properly.

If the command fails during execution, you will also need to delete the previously created .morph-holesky directory before restarting.
:::

Advanced Usage

With the Quick Start guide above, you can quickly start a node using the default configuration files. However, we also support customizing the node's home directory and parameter settings.

Customizing Data Directory
The host directory paths that are mounted by the Docker container are specified in the ops/publicnode/.env file.

```
js title="ops/publicnode/.env"
// the home folder of your Morph node
NODEHOME=${HOME}/.morph-holesky
// the data directory for your execution client: geth
GETHDATADIR=${NODEHOME}/geth-data
// the data directory for you consensus client: tendermint
NODEDATADIR=${NODEHOME}/node-data
// the entrypoint shell script for start execution client
GETHENTRYPOINTFILE=${NODEHOME}/entrypoint-geth.sh
// the jwt secret file for communicating between execution client and consensus client via engine API
JWTSECRETFILE=${NODEHOME}/jwt-secret.txt
// the snapshot name for holesky Morph node
SNAPSHOTNAME=snapshot-20240805-1
```

You have the flexibility to customize the directory paths as per your requirements.
Please note that if you want to execute make run-holesky-node to generate the necessary configuration files and snapshots for running the node, you need to ensure that the specified node home directory is new (not previously created) and do NOT alter the paths for GETHDATADIR and NODEDATADIR.

Customizing parameters

The default configuration required for node startup is located in the ops/publicnode/holesky directory. If your node home directory is empty, the run command will automatically copy these configuration files to the directory mounted in the node's docker container.

```
javascript
└── holesky
    ├── entrypoint-geth.sh
    ├── geth-data
    │   └── static-nodes.json
    └── node-data
        ├── config
        │   ├── config.toml
        │   └── genesis.json
        └── data
```

If you wish to modify the Geth startup command, you can do so by editing the entrypoint-geth.sh file. For adjustments to the Tendermint-related configuration parameters, you should modify the node-data/config/config.toml file.
Note that if you have customized your GETHDATADIR and NODEDATADIR, you will need to manually place the modified configuration files in the appropriate locations.
Managing Snapshots Yourself

You may also manually manage snapshot, particularly if you are using custom paths for the node directories.
The make download-and-decompress-snapshot command in the ops/publicnode directory will assist you in downloading and decompressing the snapshot archive.

Then, you need to manually place the decompressed data files in the appropriate node data directories.
For example, if the snapshot folder is named snapshot-20240805-1, move the contents from snapshot-20240805-1/geth to the ${GETHDATADIR}/geth directory and the contents from snapshot-20240805-1/data to the ${NODEDATADIR}/data directory.

3-how-to-run-a-validator-node.md:

---
title: How to Run a Morph Validator Node
lang: en-US

Run a Morph Node

This guide describes the approach to running a Morph validator node. If you are unfamiliar with the validator duties, please refer to our optimistic zkEVM design.

Create the folder /.morph as our home directory for this example.

Build executable binary

Clone Morph

bash
mkdir -p /.morph
cd /.morph
git clone https://github.com/morph-l2/morph.git

Currently, we use tag v0.2.0-beta as our beta version geth.

bash
cd morph
git checkout v0.2.0-beta

Build Geth

Notice: You need C compiler to build geth

bash

```
make ncccgeth
```

Build Node

```
bash
cd /.morph/morph/node
make build
```

Sync from the genesis block
Config Preparation

1. Download the config files and make data dir

```
bash
cd /.morph
wget https://raw.githubusercontent.com/morph-l2/config-template/main/holesky/
data.zip
unzip data.zip
```

2. Create a shared secret with node

```
bash
cd /.morph
openssl rand -hex 32 > jwt-secret.txt
```

Script to start the process

Geth

```
bash

NETWORKID=2810

nohup ./morph/go-ethereum/build/bin/geth \
--datadir=./geth-data \
--verbosity=3 \
--http \
--http.corsdomain="" \
--http.vhosts="" \
--http.addr=0.0.0.0 \
--http.port=8545 \
--http.api=web3,eth,txpool,net,engine \
--ws \
--ws.addr=0.0.0.0 \
--ws.port=8546 \
--ws.origins="" \
--ws.api=web3,eth,txpool,net,engine \
--networkid=$NETWORKID \
--authrpc.addr="0.0.0.0" \
--authrpc.port="8551" \
--authrpc.vhosts="" \
--authrpc.jwtsecret=$JWTSECRETPATH \
--gcmode=archive \
--metrics \
--metrics.addr=0.0.0.0 \
--metrics.port=6060 \
--miner.gasprice="100000000"
```

tail -f geth.log to check if the Geth is running properly, or you can also execute the below curl command to check if you are connected to the peer.

```
bash
curl --location --request POST 'localhost:8545/' \
--header 'Content-Type: application/json' \
--data-raw '{
   "jsonrpc":"2.0",
   "method":"ethblockNumber",
   "id":1
}'

{"jsonrpc":"2.0","id":1,"result":"0x148e39"}
```

Node

```
bash
cd /.morph
export
L1MessageQueueWithGasPriceOracle=0x778d1d9a4d8b6b9ade36d967a9ac19455ec3fd0b
export STARTHEIGHT=1434640
export Rollup=0xd8c5c541d56f59d65cf775de928ccf4a47d4985c
./morph/node/build/bin/morphnode --validator --home ./node-data \
     --l2.jwt-secret ./jwt-secret.txt \
     --l2.eth http://localhost:8545 \
     --l2.engine http://localhost:8551 \
     --l1.rpc $(Ethereum Holesky RPC)  \
     --l1.beaconrpc $(Ethereum Holesky beacon chain RPC)  \
     --l1.chain-id  17000   \
     --validator.privateKey $(Your Validator Key)  \
     --sync.depositContractAddr $(L1MessageQueueWithGasPriceOracle) \
     --sync.startHeight  $(STARTHEIGHT) \
     --derivation.rollupAddress $(Rollup) \
     --derivation.startHeight  $(STARTHEIGHT) \
     --derivation.fetchBlockRange 200 \
     --log.filename ./node.log
```

Check Status

If your node is successfully started, you will see the following response:

```
bash
I[2024-06-06|15:57:35.216] metrics server enabled
module=derivation host=0.0.0.0 port=26660
derivation node starting
ID> 24-06-06|15:57:35.216] initial sync start
module=syncer msg="Running initial sync of L1 messages before starting
sequencer, this might take a while..."
I[2024-06-06|15:57:35.242] initial sync completed
module=syncer latestSyncedBlock=1681622
I[2024-06-06|15:57:35.242] derivation start pull rollupData form l1
module=derivation startBlock=1681599 end=1681622
I[2024-06-06|15:57:35.244] fetched rollup tx
module=derivation txNum=8 latestBatchIndex=59201
I[2024-06-06|15:57:35.315] fetch rollup transaction success
module=derivation txNonce=8764
txHash=0x5fb8a98472d1be73be2bc6be0807b9e0c68b7ba14a648c8a17bdaff7b26eb923
l1BlockNumber=1681599 firstL2BlockNumber=1347115 lastL2BlockNumber=1347129
I[2024-06-06|15:57:35.669] new l2 block success
module=derivation blockNumber=1347115
```

You can use the following command to check the newest block height to make sure

you are aligned.

bash
curl --location --request POST 'localhost:8545/' \
--header 'Content-Type: application/json' \
--data-raw '{
   "jsonrpc":"2.0",
   "method":"ethblockNumber",
   "id":1
}'
{"jsonrpc":"2.0","id":1,"result":"0x148e39"}

Make sure you check the validator status constantly, if you find response

bash
[2024-06-14|16:43:50.904] root hash or withdrawal hash is not equal
originStateRootHash=0x13f91d1c272e48e2d864ce7bfb421506d5b2a04def64d45c75391cdcdd
69cd78
deriveStateRootHash=0x27e10420c0e34676a7d75c4189d7ccd1c3407cc8fd0b3eafb01c15e250
a1215f
batchWithdrawalRoot=0xa3e4a7cf45c7591a6bd9868f1fa7485ae345f10067acaade5f5b07d418
b2e172
deriveWithdrawalRoot=0xa3e4a7cf45c7591a6bd9868f1fa7485ae345f10067acaade5f5b07d41
8b2e172

This means your validators find inconsistent between sequencer submission and your own observation.

1-upgrade-node-host.md:

---
title: Upgrade node running on the host
lang: en-US

Upgrading the node is straightforward. Simply install the new version of the node executable file and replace the previous version. Then, stop the currently running node and restart it with the updated version. Node will automatically use the data of your old node and sync the latest blocks that were mined since you shut down the old software.

Running the node requires two binary files: morphnode and geth. Choose to upgrade the binary files according to your specific needs.

Step1: Compile the new version of the code

bash
git clone https://github.com/morph-l2/morph.git
checkout the latest version of the source code you need
git checkout ${latestVersion}
install geth
make ncccgeth
install morphnode
cd ./morph/node && make build

Step2: Stop nodes

bash
stop morphnode process
pid=ps -ef | grep morphnode | grep -v grep | awk '{print $2}'
kill $pid

```
stop geth process
pid=ps -ef | grep geth | grep -v grep | awk '{print $2}'
kill  $pid


Step3: Restart

Make sure to use the same start-up command you used before the upgrade

bash
start geth
./morph/go-ethereum/build/bin/geth --morph-holesky \
    --datadir "./geth-data" \
    --http --http.api=web3,debug,eth,txpool,net,engine \
    --authrpc.addr localhost \
    --authrpc.vhosts="localhost" \
    --authrpc.port 8551 \
    --authrpc.jwtsecret=./jwt-secret.txt \
    --log.filename=./geth.log

start geth
./morph/node/build/bin/morphnode --home ./node-data \
     --l2.jwt-secret ./jwt-secret.txt \
     --l2.eth http://localhost:8545 \
     --l2.engine http://localhost:8551 \
     --log.filename ./node.log


# 2-upgrade-node-docker.md:

---
title: Upgrade node running from docker
lang: en-US
---

If you are running the Docker container for the node using a custom setup, you
will need to update the docker image yourself and then restart the container.

The source code is available at https://github.com/morph-l2/morph.git. You need
to switch to the latest version of the code and then update your docker image.

If you are using  Run a Morph node with docker to start the docker container,
you can follow the subsequent steps to upgrade the node.

Step1:  Fetch latest code version
bash
git clone https://github.com/morph-l2/morph.git
checkout the latest version of the source code you need
git checkout ${latestVersion}

Step2: Stop the nodes and delete previous images

bash
stop docker container
cd ops/publicnode
make stop-holesky-node
make rm-holesky-node
delete the pervious docker image for node
docker rmi morph/node:latest
delete the pervious docker image for geth
docker rmi morph/geth-nccc:latest
```

Step3: Build the latest image and restart the container

:::note
Please note that we need to ensure that the Docker container startup parameters are consistent with those used previously. If you used a custom configuration before, make sure that the configuration and directory paths used in this run are the same as before. For details, please refer to Advanced Usage
:::

bash
start the docker container, it will automatically build the new docker images
make run-holesky-node

globals.md:

---
title: Use SDK to interact with Morph
lang: en-US
keywords: [morph,ethereum,rollup,layer2,validity proof,optimistic zk-rollup]
description: Upgrade your blockchain experience with Morph - the secure decentralized, cost0efficient, and high-performing optimistic zk-rollup solution. Try it now!

@morph-l2/sdk • Docs

@morph-l2/sdk

The @morph-l2/sdk package provides a set of tools for interacting with Morph.

Installation

npm install @morph-l2/sdk@latest

Docs

You can find auto-generated API documentation over at docs.morphl2.io.

Using the SDK

CrossChainMessenger

The CrossChainMessenger class simplifies the process of moving assets and data between Ethereum and Morph.
You can use this class to, for example, initiate a withdrawal of ERC20 tokens from Morph back to Ethereum, accurately track when the withdrawal is ready to be finalized on Ethereum, and execute the finalization transaction after the challenge period has elapsed.
The CrossChainMessenger can handle deposits and withdrawals of ETH and any ERC20-compatible token.
The CrossChainMessenger automatically connects to all relevant contracts so complex configuration is not necessary.

L2Provider and related utilities

The Morph SDK includes various utilities for handling Morph's transaction fee

model.
For instance, estimateTotalGasCost will estimate the total cost (in wei) to send at transaction on Morph including both the L2 execution cost and the L1 data cost.
You can also use the asL2Provider function to wrap an ethers Provider object into an L2Provider which will have all of these helper functions attached.

Other utilities

The SDK contains other useful helper functions and constants.
For a complete list, refer to the auto-generated SDK documentation

Documents
Enumerations

- L1ChainID
- L1RpcUrls
- L2ChainID
- L2RpcUrls
- MessageDirection
- MessageReceiptStatus
- MessageStatus

Classes

- CrossChainMessenger
- ETHBridgeAdapter
- StandardBridgeAdapter

Interfaces

- BridgeAdapterData
- BridgeAdapters
- CoreCrossChainMessage
- CrossChainMessage
- CrossChainMessageRequest
- IActionOptions
- IBridgeAdapter
- L2Block
- L2BlockWithTransactions
- L2Transaction
- MessageReceipt
- OEContracts
- OEContractsLike
- OEL1Contracts
- OEL2Contracts
- ProvenWithdrawal
- StateRoot
- StateRootBatch
- StateRootBatchHeader
- TokenBridgeMessage
- WithdrawMessageProof
- WithdrawalEntry

Type Aliases

- AddressLike
- DeepPartial
- L1Provider
- L2Provider
- LowLevelMessage
- MessageLike
- MessageRequestLike

- NumberLike
- OEL1ContractsLike
- OEL2ContractsLike
- ProviderLike
- SignerLike
- SignerOrProviderLike
- TransactionLike

Variables

- BRIDGE\ADAPTER\DATA
- CHAIN\BLOCK\TIMES
- CONTRACT\ADDRESSES
- DEFAULT\L1\CONTRACT\ADDRESSES
- DEFAULT\L2\CONTRACT\ADDRESSES
- DEPOSIT\CONFIRMATION\BLOCKS
- l1BridgeName
- l1CrossDomainMessengerName
- l2BridgeName
- l2CrossDomainMessengerName

Functions

- asL2Provider
- estimateL1Gas
- estimateL1GasCost
- estimateL2GasCost
- estimateTotalGasCost
- getAllOEContracts
- getBridgeAdapters
- getL1GasPrice
- getOEContract
- hashLowLevelMessageV2
- hashMessageHash
- isL2Provider
- migratedWithdrawalGasLimit
- omit
- toAddress
- toBigNumber
- toNumber
- toProvider
- toSignerOrProvider
- toTransactionHash

intro.md:

@morph-l2/sdk • Docs

@morph-l2/sdk

Enumerations

- L1ChainID
- L1RpcUrls
- L2ChainID
- L2RpcUrls
- MessageDirection
- MessageReceiptStatus
- MessageStatus

Classes

- CrossChainMessenger
- ETHBridgeAdapter
- StandardBridgeAdapter

Interfaces

- BridgeAdapterData
- BridgeAdapters
- CoreCrossChainMessage
- CrossChainMessage
- CrossChainMessageRequest
- IActionOptions
- IBridgeAdapter
- L2Block
- L2BlockWithTransactions
- L2Transaction
- MessageReceipt
- OEContracts
- OEContractsLike
- OEL1Contracts
- OEL2Contracts
- ProvenWithdrawal
- StateRoot
- StateRootBatch
- StateRootBatchHeader
- TokenBridgeMessage
- WithdrawMessageProof
- WithdrawalEntry

Type Aliases

- AddressLike
- DeepPartial
- L1Provider
- L2Provider
- LowLevelMessage
- MessageLike
- MessageRequestLike
- NumberLike
- OEL1ContractsLike
- OEL2ContractsLike
- ProviderLike
- SignerLike
- SignerOrProviderLike
- TransactionLike

Variables

- BRIDGE\ADAPTER\DATA
- CHAIN\BLOCK\TIMES
- CONTRACT\ADDRESSES
- DEFAULT\L1\CONTRACT\ADDRESSES
- DEFAULT\L2\CONTRACT\ADDRESSES
- DEPOSIT\CONFIRMATION\BLOCKS
- l1BridgeName
- l1CrossDomainMessengerName
- l2BridgeName
- l2CrossDomainMessengerName

Functions

- asL2Provider
- estimateL1Gas

- estimateL1GasCost
- estimateL2GasCost
- estimateTotalGasCost
- getAllOEContracts
- getBridgeAdapters
- getL1GasPrice
- getOEContract
- hashLowLevelMessageV2
- hashMessageHash
- isL2Provider
- migratedWithdrawalGasLimit
- omit
- toAddress
- toBigNumber
- toNumber
- toProvider
- toSignerOrProvider
- toTransactionHash

CrossChainMessenger.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / CrossChainMessenger

Class: CrossChainMessenger

Constructors

new CrossChainMessenger()

> new CrossChainMessenger(opts): CrossChainMessenger

Creates a new CrossChainProvider instance.

Parameters

• opts

Options for the provider.

• opts.backendURL?: string

backend for withdraw proof gen.

• opts.bridges?: BridgeAdapterData

Optional bridge address list.

• opts.contracts?: DeepPartial\<OEContractsLike\>

Optional contract address overrides.

• opts.l1ChainId: NumberLike

Chain ID for the L1 chain.

• opts.l1SignerOrProvider: SignerOrProviderLike

Signer or Provider for the L1 chain, or a JSON-RPC url.

• opts.l2ChainId: NumberLike

Chain ID for the L2 chain.

• opts.l2SignerOrProvider: SignerOrProviderLike

Signer or Provider for the L2 chain, or a JSON-RPC url.

Returns

CrossChainMessenger

Source

src/cross-chain-messenger.ts:130

Properties

backendURL

> backendURL: string

Backend url for withdrawal prove server

Source

src/cross-chain-messenger.ts:76

bridges

> bridges: BridgeAdapters

List of custom bridges for the given network.

Source

src/cross-chain-messenger.ts:116

contracts

> contracts: OEContracts

Contract objects attached to their respective providers and addresses.

Source

src/cross-chain-messenger.ts:111

estimateGas

> estimateGas: object

Object that holds the functions that estimates the gas required for a given transaction.
Follows the pattern used by ethers.js.

approveERC20()

> approveERC20: (l1Token, l2Token, amount, opts?) => Promise\<BigNumber\>

Estimates gas required to approve some tokens to deposit into the L2 chain.

Parameters

• l1Token: AddressLike

The L1 token address.

• l2Token: AddressLike

The L2 token address.

• amount: NumberLike

Amount of the token to approve.

• opts?: IActionOptions

Additional options.

Returns

Promise\<BigNumber\>

depositERC20()

> depositERC20: (l1Token, l2Token, amount, opts?) => Promise\<BigNumber\>

Estimates gas required to deposit some ERC20 tokens into the L2 chain.

Parameters

• l1Token: AddressLike

Address of the L1 token.

• l2Token: AddressLike

Address of the L2 token.

• amount: NumberLike

Amount to deposit.

• opts?: IActionOptions

Additional options.

Returns

Promise\<BigNumber\>

depositETH()

> depositETH: (amount, opts?) => Promise\<BigNumber\>

Estimates gas required to deposit some ETH into the L2 chain.

Parameters

• amount: NumberLike

Amount of ETH to deposit.

• opts?: IActionOptions

Additional options.

Returns

Promise\<BigNumber\>

proveAndRelayMessage()

> proveAndRelayMessage: (message, opts?) => Promise\<BigNumber\>

Estimates gas required to proveAndRelay a cross chain message. Only applies to L2 to L1 messages.

Parameters

• message: MessageLike

Message to generate the proving transaction for.

• opts?: IActionOptions

Additional options.

Returns

Promise\<BigNumber\>

sendMessage()

> sendMessage: (message, opts?) => Promise\<BigNumber\>

Estimates gas required to send a cross chain message.

Parameters

• message: CrossChainMessageRequest

Cross chain message to send.

• opts?: IActionOptions

Additional options.

Returns

Promise\<BigNumber\>

withdrawERC20()

> withdrawERC20: (l1Token, l2Token, amount, opts?) => Promise\<BigNumber\>

Estimates gas required to withdraw some ERC20 tokens back to the L1 chain.

Parameters

• l1Token: AddressLike

Address of the L1 token.

• l2Token: AddressLike

Address of the L2 token.

• amount: NumberLike

Amount to withdraw.

• opts?: IActionOptions

Additional options.

Returns

Promise\<BigNumber\>

withdrawETH()

> withdrawETH: (amount, opts?) => Promise\<BigNumber\>

Estimates gas required to withdraw some ETH back to the L1 chain.

Parameters

• amount: NumberLike

Amount of ETH to withdraw.

• opts?: IActionOptions

Additional options.

Returns

Promise\<BigNumber\>

Source

src/cross-chain-messenger.ts:1600

l1ChainId

> l1ChainId: number

Chain ID for the L1 network.

Source

src/cross-chain-messenger.ts:101

l1CrossDomainMessenger

> l1CrossDomainMessenger: Contract

CrossDomainMessenger connected to the L1 chain.

Source

src/cross-chain-messenger.ts:91

l1SignerOrProvider

> l1SignerOrProvider: Provider \| Signer

Provider connected to the L1 chain.

Source

src/cross-chain-messenger.ts:81

l2ChainId

> l2ChainId: number

Chain ID for the L2 network.

Source

src/cross-chain-messenger.ts:106

l2CrossDomainMessenger

> l2CrossDomainMessenger: Contract

CrossDomainMessenger connected to the L2 chain.

Source

src/cross-chain-messenger.ts:96

l2SignerOrProvider

> l2SignerOrProvider: Provider \| Signer

Provider connected to the L2 chain.

Source

src/cross-chain-messenger.ts:86

populateTransaction

> populateTransaction: object

Object that holds the functions that generate transactions to be signed by the user.
Follows the pattern used by ethers.js.

approveERC20()

> approveERC20: (l1Token, l2Token, amount, opts?) => Promise\
<TransactionRequest\>

Generates a transaction for approving some tokens to deposit into the L2 chain.

Parameters

• l1Token: AddressLike

The L1 token address.

• l2Token: AddressLike

The L2 token address.

• amount: NumberLike

Amount of the token to approve.

• opts?: IActionOptions

Additional options.

Returns

Promise\<TransactionRequest\>

depositERC20()

> depositERC20: (l1Token, l2Token, amount, opts?, isEstimatingGas) => Promise\<TransactionRequest\>

Generates a transaction for depositing some ERC20 tokens into the L2 chain.

Parameters

• l1Token: AddressLike

Address of the L1 token.

• l2Token: AddressLike

Address of the L2 token.

• amount: NumberLike

Amount to deposit.

• opts?: IActionOptions

Additional options.

• isEstimatingGas?: boolean= false

Returns

Promise\<TransactionRequest\>

depositETH()

> depositETH: (amount, opts?, isEstimatingGas) => Promise\<TransactionRequest\>

Generates a transaction for depositing some ETH into the L2 chain.

Parameters

• amount: NumberLike

Amount of ETH to deposit.

• opts?: IActionOptions

Additional options.

• isEstimatingGas?: boolean= false

Returns

Promise\<TransactionRequest\>

proveAndRelayMessage()

> proveAndRelayMessage: (message, opts?) => Promise\<TransactionRequest\>

Generates a message proving and relaying transaction that can be signed and executed. Only
applicable for L2 to L1 messages.

Parameters

• message: MessageLike

Message to generate the proving transaction for.

• opts?: IActionOptions

Additional options.

Returns

Promise\<TransactionRequest\>

sendMessage()

> sendMessage: (message, opts?) => Promise\<TransactionRequest\>

Generates a transaction that sends a given cross chain message. This transaction can be signed
and executed by a signer.

Parameters

• message: CrossChainMessageRequest

Cross chain message to send.

• opts?: IActionOptions

Additional options.

Returns

Promise\<TransactionRequest\>

withdrawERC20()

> withdrawERC20: (l1Token, l2Token, amount, opts?, isEstimatingGas?) => Promise\<TransactionRequest\>

Generates a transaction for withdrawing some ERC20 tokens back to the L1 chain.

Parameters

• l1Token: AddressLike

Address of the L1 token.

• l2Token: AddressLike

Address of the L2 token.

• amount: NumberLike

Amount to withdraw.

• opts?: IActionOptions

Additional options.

• isEstimatingGas?: boolean

Returns

Promise\<TransactionRequest\>

withdrawETH()

> withdrawETH: (amount, opts?, isEstimatingGas?) => Promise\<TransactionRequest\
>

Generates a transaction for withdrawing some ETH back to the L1 chain.

Parameters

• amount: NumberLike

Amount of ETH to withdraw.

• opts?: IActionOptions

Additional options.

• isEstimatingGas?: boolean

Returns

Promise\<TransactionRequest\>

Source

src/cross-chain-messenger.ts:1304

Accessors

l1Provider

> get l1Provider(): Provider

Provider connected to the L1 chain.

Returns

Provider

Source

src/cross-chain-messenger.ts:193

l1Signer

> get l1Signer(): Signer

Signer connected to the L1 chain.

Returns

Signer

Source

src/cross-chain-messenger.ts:215

l2Provider

> get l2Provider(): Provider

Provider connected to the L2 chain.

Returns

Provider

Source

src/cross-chain-messenger.ts:204

l2Signer

> get l2Signer(): Signer

Signer connected to the L2 chain.

Returns

Signer

Source

src/cross-chain-messenger.ts:226

Methods

approval()

> approval(l1Token, l2Token, opts?): Promise\<BigNumber\>

Queries the account's approval amount for a given L1 token.

Parameters

• l1Token: AddressLike

The L1 token address.

• l2Token: AddressLike

The L2 token address.

• opts?: IActionOptions

Additional options.

Returns

Promise\<BigNumber\>

Amount of tokens approved for deposits from the account.

Source

src/cross-chain-messenger.ts:1214

approveERC20()

> approveERC20(l1Token, l2Token, amount, opts?): Promise\<TransactionResponse\>

Approves a deposit into the L2 chain.

Parameters

• l1Token: AddressLike

The L1 token address.

• l2Token: AddressLike

The L2 token address.

• amount: NumberLike

Amount of the token to approve.

• opts?: IActionOptions

Additional options.

Returns

Promise\<TransactionResponse\>

Transaction response for the approval transaction.

Source

src/cross-chain-messenger.ts:1233

depositERC20()

> depositERC20(l1Token, l2Token, amount, opts?): Promise\<TransactionResponse\>

Deposits some ERC20 tokens into the L2 chain.

Parameters

• l1Token: AddressLike

Address of the L1 token.

• l2Token: AddressLike

Address of the L2 token.

• amount: NumberLike

Amount to deposit.

• opts?: IActionOptions

Additional options.

Returns

Promise\<TransactionResponse\>

Transaction response for the deposit transaction.

Source

src/cross-chain-messenger.ts:1256

depositETH()

> depositETH(amount, opts?): Promise\<TransactionResponse\>

Deposits some ETH into the L2 chain.

Parameters

• amount: NumberLike

Amount of ETH to deposit (in wei).

• opts?: IActionOptions

Additional options.

Returns

Promise\<TransactionResponse\>

Transaction response for the deposit transaction.

Source

src/cross-chain-messenger.ts:1183

encodeFunctionMessage()

> encodeFunctionMessage(options): string

L2CrossDomainMessenger contract encode message, such as hashCrossDomainMessagev1

Parameters

• options

• options.message: string

The message passed along with the cross domain message

• options.messageNonce: BigNumber

The cross domain message nonce

• options.sender: string

The sender of the cross domain message

• options.target: string

The target of the cross domain message

• options.value: BigNumber

The value being sent with the cross domain message

Returns

string

Source

src/cross-chain-messenger.ts:972

estimateL2MessageGasLimit()

> estimateL2MessageGasLimit(message, opts?): Promise\<BigNumber\>

Estimates the amount of gas required to fully execute a given message on L2. Only applies to
L1 => L2 messages. You would supply this gas limit when sending the message to L2.

Parameters

• message: MessageRequestLike

Message get a gas estimate for.

• opts?

Options object.

• opts.bufferPercent?: number

Percentage of gas to add to the estimate. Defaults to 20.

• opts.from?: string

Address to use as the sender.

Returns

Promise\<BigNumber\>

Estimates L2 gas limit.

Source

src/cross-chain-messenger.ts:918

getBackendTreeSyncIndex()

> getBackendTreeSyncIndex(): Promise\<number\>

Returns

Promise\<number\>

Source

src/cross-chain-messenger.ts:1122

getBridgeForTokenPair()

> getBridgeForTokenPair(l1Token, l2Token): Promise\<IBridgeAdapter\>

Finds the appropriate bridge adapter for a given L1 - L2 token pair. Will throw if no bridges
support the token pair or if more than one bridge supports the token pair.

Parameters

• l1Token: AddressLike

L1 token address.

• l2Token: AddressLike

L2 token address.

Returns

Promise\<IBridgeAdapter\>

The appropriate bridge adapter for the given token pair.

Source

src/cross-chain-messenger.ts:378

getCommittedL2BlockNumber()

> getCommittedL2BlockNumber(): Promise\<any\>

Returns

Promise\<any\>

Source

src/cross-chain-messenger.ts:995

getDepositsByAddress()

> getDepositsByAddress(address, opts): Promise\<TokenBridgeMessage[]\>

Gets all deposits for a given address.

Parameters

• address: AddressLike

Address to search for messages from.

• opts= {}

Options object.

• opts.fromBlock?: BlockTag

Block to start searching for messages from. If not provided, will start from the first block (block #0).

• opts.toBlock?: BlockTag

Block to stop searching for messages at. If not provided, will stop at the latest known block ("latest").

Returns

Promise\<TokenBridgeMessage[]\>

All deposit token bridge messages sent by the given address.

Source

src/cross-chain-messenger.ts:424

getFinalizedL2BlockNumber()

> getFinalizedL2BlockNumber(): Promise\<any\>

Returns

Promise\<any\>

Source

src/cross-chain-messenger.ts:1017

getMessageReceipt()

> getMessageReceipt(message, opts): Promise\<MessageReceipt\>

Finds the receipt of the transaction that executed a particular cross chain message.

Parameters

• message: MessageLike

Message to find the receipt of.

• opts= {}

• opts.direction?: MessageDirection

Returns

Promise\<MessageReceipt\>

CrossChainMessage receipt including receipt of the transaction that relayed the given message.

Source

src/cross-chain-messenger.ts:757

getMessageStatus()

> getMessageStatus(message, opts): Promise\<MessageStatus\>

Retrieves the status of a particular message as an enum.

Parameters

• message: MessageLike

Cross chain message to check the status of.

• opts= {}

• opts.direction?: MessageDirection

Returns

Promise\<MessageStatus\>

Status of the message.

Source

src/cross-chain-messenger.ts:634

getMessagesByTransaction()

> getMessagesByTransaction(transaction, opts): Promise\<CrossChainMessage[]\>

Retrieves all cross chain messages sent within a given transaction.

Parameters

• transaction: TransactionLike

Transaction hash or receipt to find messages from.

• opts= {}

Options object.

• opts.direction?: MessageDirection

Direction to search for messages in. If not provided, will attempt to automatically search both directions under the assumption that a transaction hash will only
exist on one chain. If the hash exists on both chains, will throw an error.

Returns

Promise\<CrossChainMessage[]\>

All cross chain messages sent within the transaction.

Source

src/cross-chain-messenger.ts:244

getProvenWithdrawal()

> getProvenWithdrawal(withdrawalHash): Promise\<ProvenWithdrawal\>

Queries the L1CrossDomainMessenger contract's provenWithdrawals mapping for a ProvenWithdrawal that matches the passed withdrawalHash

Parameters

• withdrawalHash: string

Returns

Promise\<ProvenWithdrawal\>

A ProvenWithdrawal object

Source

src/cross-chain-messenger.ts:957

getWithdrawMessageProof()

> getWithdrawMessageProof(message): Promise\<WithdrawMessageProof\>

Generates the proof required to finalize an L2 to L1 message.

Parameters

• message: MessageLike

Message to generate a proof for.

Returns

Promise\<WithdrawMessageProof\>

Proof that can be used to finalize the message.

Source

src/cross-chain-messenger.ts:1042

getWithdrawalsByAddress()

> getWithdrawalsByAddress(address, opts): Promise\<TokenBridgeMessage[]\>

Gets all withdrawals for a given address.

Parameters

• address: AddressLike

Address to search for messages from.

• opts= {}

Options object.

• opts.fromBlock?: BlockTag

Block to start searching for messages from. If not provided, will start from the first block (block #0).

• opts.toBlock?: BlockTag

Block to stop searching for messages at. If not provided, will stop at the latest known block ("latest").

Returns

Promise\<TokenBridgeMessage[]\>

All withdrawal token bridge messages sent by the given address.

Source

src/cross-chain-messenger.ts:458

proveAndRelayMessage()

> proveAndRelayMessage(message, opts?): Promise\<TransactionResponse\>

Prove and relay a cross chain message that was sent from L2 to L1. Only applicable for L2 to L1
messages.

Parameters

• message: MessageLike

Message to finalize.

• opts?: IActionOptions

Additional options.

Returns

Promise\<TransactionResponse\>

Transaction response for the finalization transaction.

Source

src/cross-chain-messenger.ts:1163

sendMessage()

> sendMessage(message, opts?): Promise\<TransactionResponse\>

Sends a given cross chain message. Where the message is sent depends on the direction attached
to the message itself.

Parameters

• message: CrossChainMessageRequest

Cross chain message to send.

• opts?: IActionOptions

Additional options.

Returns

Promise\<TransactionResponse\>

Transaction response for the message sending transaction.

Source

src/cross-chain-messenger.ts:1143

toCrossChainMessage()

> toCrossChainMessage(message, opts?): Promise\<CrossChainMessage\>

Resolves a MessageLike into a CrossChainMessage object.
Unlike other coercion functions, this function is stateful and requires making additional
requests. For now I'm going to keep this function here, but we could consider putting a
similar function inside of utils/coercion.ts if people want to use this without having to
create an entire CrossChainProvider object.

Parameters

• message: MessageLike

MessageLike to resolve into a CrossChainMessage.

• opts?

• opts.direction?: MessageDirection

Returns

Promise\<CrossChainMessage\>

Message coerced into a CrossChainMessage.

Source

src/cross-chain-messenger.ts:491

toLowLevelMessage()

> toLowLevelMessage(message, opts?): Promise\<LowLevelMessage\>

Transforms a CrossChainMessenger message into its low-level representation inside the
L2ToL1MessagePasser contract on L2.

Parameters

• message: MessageLike

Message to transform.

• opts?

• opts.direction?: MessageDirection

Returns

Promise\<LowLevelMessage\>

Transformed message.

Source

src/cross-chain-messenger.ts:326

waitBatchFinalize()

> waitBatchFinalize(transactionHash): Promise\<void\>

Parameters

• transactionHash: string

Returns

Promise\<void\>

Source

src/cross-chain-messenger.ts:600

waitForMessageReceipt()

> waitForMessageReceipt(message, opts): Promise\<MessageReceipt\>

Waits for a message to be executed and returns the receipt of the transaction that executed
the given message.

Parameters

• message: MessageLike

Message to wait for.

• opts= {}

Options to pass to the waiting function.

• opts.confirmations?: number

Number of transaction confirmations to wait for before returning.

• opts.pollIntervalMs?: number

Number of milliseconds to wait between polling for the receipt.

• opts.timeoutMs?: number

Milliseconds to wait before timing out.

Returns

Promise\<MessageReceipt\>

CrossChainMessage receipt including receipt of the transaction that relayed the given message.

Source

src/cross-chain-messenger.ts:802

waitForMessageStatus()

> waitForMessageStatus(message, status, opts): Promise\<void\>

Waits until the status of a given message changes to the expected status. Note that if the
status of the given message changes to a status that implies the expected status, this will
still return. If the status of the message changes to a status that exclues the expected
status, this will throw an error.

Parameters

• message: MessageLike

Message to wait for.

• status: MessageStatus

Expected status of the message.

• opts= {}

Options to pass to the waiting function.

• opts.direction?: MessageDirection

• opts.pollIntervalMs?: number

Number of milliseconds to wait when polling.

• opts.timeoutMs?: number

Milliseconds to wait before timing out.

Returns

Promise\<void\>

Source

src/cross-chain-messenger.ts:840

waitRollupSuccess()

> waitRollupSuccess(transactionHash): Promise\<void\>

Parameters

• transactionHash: string

Returns

Promise\<void\>

Source

src/cross-chain-messenger.ts:552

waitSyncSuccess()

> waitSyncSuccess(transactionHash): Promise\<void\>

Parameters

• transactionHash: string

Returns

Promise\<void\>

Source

src/cross-chain-messenger.ts:576

withdrawERC20()

> withdrawERC20(l1Token, l2Token, amount, opts?): Promise\<TransactionResponse\>

Withdraws some ERC20 tokens back to the L1 chain.

Parameters

• l1Token: AddressLike

Address of the L1 token.

• l2Token: AddressLike

Address of the L2 token.

• amount: NumberLike

Amount to withdraw.

• opts?: IActionOptions

Additional options.

Returns

Promise\<TransactionResponse\>

Transaction response for the withdraw transaction.

Source

src/cross-chain-messenger.ts:1282

withdrawETH()

> withdrawETH(amount, opts?): Promise\<TransactionResponse\>

Withdraws some ETH back to the L1 chain.

Parameters

• amount: NumberLike

Amount of ETH to withdraw.

• opts?: IActionOptions

Additional options.

Returns

Promise\<TransactionResponse\>

Transaction response for the withdraw transaction.

Source

src/cross-chain-messenger.ts:1198

ETHBridgeAdapter.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / ETHBridgeAdapter

Class: ETHBridgeAdapter

Bridge adapter for the ETH bridge.

Extends

- StandardBridgeAdapter

Constructors

new ETHBridgeAdapter()

> new ETHBridgeAdapter(opts): ETHBridgeAdapter

Creates a StandardBridgeAdapter instance.

Parameters

• opts

Options for the adapter.

• opts.l1Bridge: AddressLike

L1 bridge contract.

• opts.l2Bridge: AddressLike

L2 bridge contract.

• opts.messenger: CrossChainMessenger

Provider used to make queries related to cross-chain interactions.

Returns

ETHBridgeAdapter

Inherited from

StandardBridgeAdapter.constructor

Source

src/adapters/standard-bridge.ts:52

Properties

estimateGas

> estimateGas: object

Object that holds the functions that estimates the gas required for a given transaction.
Follows the pattern used by ethers.js.

approve()

> approve: (l1Token, l2Token, amount, opts?) => Promise\<BigNumber\>

Parameters

• l1Token: AddressLike

• l2Token: AddressLike

• amount: NumberLike

• opts?: IActionOptions

Returns

Promise\<BigNumber\>

deposit()

> deposit: (l1Token, l2Token, amount, opts?) => Promise\<BigNumber\>

Parameters

• l1Token: AddressLike

• l2Token: AddressLike

• amount: NumberLike

• opts?: IActionOptions

Returns

Promise\<BigNumber\>

withdraw()

> withdraw: (l1Token, l2Token, amount, opts?) => Promise\<BigNumber\>

Parameters

• l1Token: AddressLike

• l2Token: AddressLike

• amount: NumberLike

• opts?: IActionOptions

Returns

Promise\<BigNumber\>

Inherited from

StandardBridgeAdapter.estimateGas

Source

src/adapters/standard-bridge.ts:405

l1Bridge

> l1Bridge: Contract

L1 bridge contract.

Inherited from

StandardBridgeAdapter.l1Bridge

Source

src/adapters/standard-bridge.ts:41

l2Bridge

> l2Bridge: Contract

L2 bridge contract.

Inherited from

StandardBridgeAdapter.l2Bridge

Source

src/adapters/standard-bridge.ts:42

messenger

> messenger: CrossChainMessenger

Provider used to make queries related to cross-chain interactions.

Inherited from

StandardBridgeAdapter.messenger

Source

src/adapters/standard-bridge.ts:40

populateTransaction

> populateTransaction: object

Object that holds the functions that generate transactions to be signed by the user.
Follows the pattern used by ethers.js.

approve()

> approve: (l1Token, l2Token, amount, opts?) => Promise\<never\>

Parameters

• l1Token: AddressLike

• l2Token: AddressLike

• amount: NumberLike

• opts?: IActionOptions

Returns

Promise\<never\>

deposit()

> deposit: (l1Token, l2Token, amount, opts) => Promise\<TransactionRequest\>

Parameters

• l1Token: AddressLike

• l2Token: AddressLike

• amount: NumberLike

• opts: IActionOptions

Returns

Promise\<TransactionRequest\>

withdraw()

> withdraw: (l1Token, l2Token, amount, opts) => Promise\<TransactionRequest\>

Parameters

• l1Token: AddressLike

• l2Token: AddressLike

• amount: NumberLike

• opts: IActionOptions

Returns

Promise\<TransactionRequest\>

Overrides

StandardBridgeAdapter.populateTransaction

Source

src/adapters/eth-bridge.ts:116

Methods

approval()

> approval(l1Token, l2Token, opts?): Promise\<BigNumber\>

Queries the account's approval amount for a given L1 token.

Parameters

• l1Token: AddressLike

The L1 token address.

• l2Token: AddressLike

The L2 token address.

• opts?: IActionOptions

Additional options.

Returns

Promise\<BigNumber\>

Amount of tokens approved for deposits from the account.

Overrides

StandardBridgeAdapter.approval

Source

src/adapters/eth-bridge.ts:22

approve()

> approve(l1Token, l2Token, amount, signer, opts?): Promise\
<TransactionResponse\>

Approves a deposit into the L2 chain.

Parameters

• l1Token: AddressLike

The L1 token address.

• l2Token: AddressLike

The L2 token address.

• amount: NumberLike

Amount of the token to approve.

• signer: Signer

Signer used to sign and send the transaction.

• opts?: IActionOptions

Additional options.

Returns

Promise\<TransactionResponse\>

Transaction response for the approval transaction.

Inherited from

StandardBridgeAdapter.approve

Source

src/adapters/standard-bridge.ts:250

deposit()

> deposit(l1Token, l2Token, amount, signer, opts?): Promise\
<TransactionResponse\>

Deposits some tokens into the L2 chain.

Parameters

• l1Token: AddressLike

The L1 token address.

• l2Token: AddressLike

The L2 token address.

• amount: NumberLike

Amount of the token to deposit.

• signer: Signer

Signer used to sign and send the transaction.

• opts?: IActionOptions

Additional options.

Returns

Promise\<TransactionResponse\>

Transaction response for the deposit transaction.

Inherited from

StandardBridgeAdapter.deposit

Source

src/adapters/standard-bridge.ts:262

getDepositsByAddress()

> getDepositsByAddress(address, opts?): Promise\<TokenBridgeMessage[]\>

Gets all deposits for a given address.

Parameters

• address: AddressLike

Address to search for messages from.

• opts?

Options object.

• opts.fromBlock?: BlockTag

• opts.toBlock?: BlockTag

Returns

Promise\<TokenBridgeMessage[]\>

All deposit token bridge messages sent by the given address.

Overrides

StandardBridgeAdapter.getDepositsByAddress

Source

src/adapters/eth-bridge.ts:30

getWithdrawalsByAddress()

> getWithdrawalsByAddress(address, opts?): Promise\<TokenBridgeMessage[]\>

Gets all withdrawals for a given address.

Parameters

• address: AddressLike

Address to search for messages from.

• opts?

Options object.

• opts.fromBlock?: BlockTag

• opts.toBlock?: BlockTag

Returns

Promise\<TokenBridgeMessage[]\>

All withdrawal token bridge messages sent by the given address.

Overrides

StandardBridgeAdapter.getWithdrawalsByAddress

Source

src/adapters/eth-bridge.ts:64

supportsTokenPair()

> supportsTokenPair(l1Token, l2Token): Promise\<boolean\>

Checks whether the given token pair is supported by the bridge.

Parameters

• l1Token: AddressLike

The L1 token address.

• l2Token: AddressLike

The L2 token address.

Returns

Promise\<boolean\>

Whether the given token pair is supported by the bridge.

Overrides

StandardBridgeAdapter.supportsTokenPair

Source

src/adapters/eth-bridge.ts:105

withdraw()

> withdraw(l1Token, l2Token, amount, signer, opts?): Promise\
<TransactionResponse\>

Withdraws some tokens back to the L1 chain.

Parameters

• l1Token: AddressLike

The L1 token address.

• l2Token: AddressLike

The L2 token address.

• amount: NumberLike

Amount of the token to withdraw.

• signer: Signer

Signer used to sign and send the transaction.

• opts?: IActionOptions

Additional options.

Returns

Promise\<TransactionResponse\>

Transaction response for the withdraw transaction.

Inherited from

StandardBridgeAdapter.withdraw

Source

src/adapters/standard-bridge.ts:274

StandardBridgeAdapter.md:

@morph-l2/sdk / StandardBridgeAdapter

Class: StandardBridgeAdapter

Bridge adapter for any token bridge that uses the standard token bridge interface.

Extended by

- ETHBridgeAdapter

Implements

- IBridgeAdapter

Constructors

new StandardBridgeAdapter()

> new StandardBridgeAdapter(opts): StandardBridgeAdapter

Creates a StandardBridgeAdapter instance.

Parameters

• opts

Options for the adapter.

• opts.l1Bridge: AddressLike

L1 bridge contract.

• opts.l2Bridge: AddressLike

L2 bridge contract.

• opts.messenger: CrossChainMessenger

Provider used to make queries related to cross-chain interactions.

Returns

StandardBridgeAdapter

Source

src/adapters/standard-bridge.ts:52

Properties

estimateGas

> estimateGas: object

Object that holds the functions that estimates the gas required for a given transaction.
Follows the pattern used by ethers.js.

approve()

> approve: (l1Token, l2Token, amount, opts?) => Promise\<BigNumber\>

Parameters

• l1Token: AddressLike

• l2Token: AddressLike

• amount: NumberLike

• opts?: IActionOptions

Returns

Promise\<BigNumber\>

deposit()

> deposit: (l1Token, l2Token, amount, opts?) => Promise\<BigNumber\>

Parameters

• l1Token: AddressLike

• l2Token: AddressLike

• amount: NumberLike

• opts?: IActionOptions

Returns

Promise\<BigNumber\>

withdraw()

> withdraw: (l1Token, l2Token, amount, opts?) => Promise\<BigNumber\>

Parameters

• l1Token: AddressLike

• l2Token: AddressLike

• amount: NumberLike

• opts?: IActionOptions

Returns

Promise\<BigNumber\>

Implementation of

IBridgeAdapter.estimateGas

Source

src/adapters/standard-bridge.ts:405

l1Bridge

> l1Bridge: Contract

L1 bridge contract.

Implementation of

IBridgeAdapter.l1Bridge

Source

src/adapters/standard-bridge.ts:41

l2Bridge

> l2Bridge: Contract

L2 bridge contract.

Implementation of

IBridgeAdapter.l2Bridge

Source

src/adapters/standard-bridge.ts:42

messenger

> messenger: CrossChainMessenger

Provider used to make queries related to cross-chain interactions.

Implementation of

IBridgeAdapter.messenger

Source

src/adapters/standard-bridge.ts:40

populateTransaction

> populateTransaction: object

Object that holds the functions that generate transactions to be signed by the user.
Follows the pattern used by ethers.js.

approve()

> approve: (l1Token, l2Token, amount, opts?) => Promise\<TransactionRequest\>

Parameters

• l1Token: AddressLike

• l2Token: AddressLike

• amount: NumberLike

• opts?: IActionOptions

Returns

Promise\<TransactionRequest\>

deposit()

> deposit: (l1Token, l2Token, amount, opts?) => Promise\<TransactionRequest\>

Parameters

• l1Token: AddressLike

• l2Token: AddressLike

• amount: NumberLike

• opts?: IActionOptions

Returns

Promise\<TransactionRequest\>

withdraw()

> withdraw: (l1Token, l2Token, amount, opts?) => Promise\<TransactionRequest\>

Parameters

• l1Token: AddressLike

• l2Token: AddressLike

• amount: NumberLike

• opts?: IActionOptions

Returns

Promise\<TransactionRequest\>

Implementation of

IBridgeAdapter.populateTransaction

Source

src/adapters/standard-bridge.ts:286

Methods

approval()

> approval(l1Token, l2Token, opts): Promise\<BigNumber\>

Queries the account's approval amount for a given L1 token.

Parameters

• l1Token: AddressLike

The L1 token address.

• l2Token: AddressLike

The L2 token address.

• opts: IActionOptions

Additional options.

Returns

Promise\<BigNumber\>

Amount of tokens approved for deposits from the account.

Implementation of

IBridgeAdapter.approval

Source

src/adapters/standard-bridge.ts:209

approve()

> approve(l1Token, l2Token, amount, signer, opts?): Promise\
<TransactionResponse\>

Approves a deposit into the L2 chain.

Parameters

• l1Token: AddressLike

The L1 token address.

• l2Token: AddressLike

The L2 token address.

• amount: NumberLike

Amount of the token to approve.

• signer: Signer

Signer used to sign and send the transaction.

• opts?: IActionOptions

Additional options.

Returns

Promise\<TransactionResponse\>

Transaction response for the approval transaction.

Implementation of

IBridgeAdapter.approve

Source

src/adapters/standard-bridge.ts:250

deposit()

> deposit(l1Token, l2Token, amount, signer, opts?): Promise\
<TransactionResponse\>

Deposits some tokens into the L2 chain.

Parameters

• l1Token: AddressLike

The L1 token address.

• l2Token: AddressLike

The L2 token address.

• amount: NumberLike

Amount of the token to deposit.

• signer: Signer

Signer used to sign and send the transaction.

• opts?: IActionOptions

Additional options.

Returns

Promise\<TransactionResponse\>

Transaction response for the deposit transaction.

Implementation of

IBridgeAdapter.deposit

Source

src/adapters/standard-bridge.ts:262

getDepositsByAddress()

> getDepositsByAddress(address, opts?): Promise\<TokenBridgeMessage[]\>

Gets all deposits for a given address.

Parameters

• address: AddressLike

Address to search for messages from.

• opts?

Options object.

• opts.fromBlock?: BlockTag

• opts.toBlock?: BlockTag

Returns

Promise\<TokenBridgeMessage[]\>

All deposit token bridge messages sent by the given address.

Implementation of

IBridgeAdapter.getDepositsByAddress

Source

src/adapters/standard-bridge.ts:75

getWithdrawalsByAddress()

> getWithdrawalsByAddress(address, opts?): Promise\<TokenBridgeMessage[]\>

Gets all withdrawals for a given address.

Parameters

• address: AddressLike

Address to search for messages from.

• opts?

Options object.

• opts.fromBlock?: BlockTag

• opts.toBlock?: BlockTag

Returns

Promise\<TokenBridgeMessage[]\>

All withdrawal token bridge messages sent by the given address.

Implementation of

IBridgeAdapter.getWithdrawalsByAddress

Source

src/adapters/standard-bridge.ts:122

supportsTokenPair()

> supportsTokenPair(l1Token, l2Token): Promise\<boolean\>

Checks whether the given token pair is supported by the bridge.

Parameters

• l1Token: AddressLike

The L1 token address.

• l2Token: AddressLike

The L2 token address.

Returns

Promise\<boolean\>

Whether the given token pair is supported by the bridge.

Implementation of

IBridgeAdapter.supportsTokenPair

Source

src/adapters/standard-bridge.ts:165

withdraw()

> withdraw(l1Token, l2Token, amount, signer, opts?): Promise\
<TransactionResponse\>

Withdraws some tokens back to the L1 chain.

Parameters

• l1Token: AddressLike

The L1 token address.

• l2Token: AddressLike

The L2 token address.

• amount: NumberLike

Amount of the token to withdraw.

• signer: Signer

Signer used to sign and send the transaction.

• opts?: IActionOptions

Additional options.

Returns

Promise\<TransactionResponse\>

Transaction response for the withdraw transaction.

Implementation of

IBridgeAdapter.withdraw

Source

src/adapters/standard-bridge.ts:274

L1ChainID.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / L1ChainID

Enumeration: L1ChainID

L1 network chain IDs

Enumeration Members

HOLESKY

> HOLESKY: 17000

Source

src/interfaces/types.ts:23

MAINNET

> MAINNET: 1

Source

src/interfaces/types.ts:17

MORPH\LOCAL\DEVNET

> MORPH\LOCAL\DEVNET: 900

Source

src/interfaces/types.ts:20

MORPH\QANET

> MORPH\QANET: 900

Source

src/interfaces/types.ts:21

SEPOLIA

> SEPOLIA: 11155111

Source

src/interfaces/types.ts:22

L1RpcUrls.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / L1RpcUrls

Enumeration: L1RpcUrls

Enumeration Members

HOLESKY

> HOLESKY: "https://1rpc.io/holesky"

Source

src/interfaces/types.ts:41

MORPH\LOCAL\DEVNET

> MORPH\LOCAL\DEVNET: "http://localhost:9545"

Source

src/interfaces/types.ts:39

SEPOLIA

> SEPOLIA: "https://1rpc.io/sepolia"

Source

src/interfaces/types.ts:40

L2ChainID.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / L2ChainID

Enumeration: L2ChainID

L2 network chain IDs

Enumeration Members

MORPH\HOLESKY

> MORPH\HOLESKY: 2810

Source

src/interfaces/types.ts:35

MORPH\LOCAL\DEVNET

> MORPH\LOCAL\DEVNET: 53077

Source

src/interfaces/types.ts:32

MORPH\MAINNET

> MORPH\MAINNET: 0

Source

src/interfaces/types.ts:30

MORPH\QANET

> MORPH\QANET: 53077

Source

src/interfaces/types.ts:33

MORPH\TESTNET

> MORPH\TESTNET: 2710

Source

src/interfaces/types.ts:34

L2RpcUrls.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / L2RpcUrls

Enumeration: L2RpcUrls

Enumeration Members

MORPH\HOLESKY

> MORPH\HOLESKY: "https://rpc-holesky.morphl2.io"

Source

src/interfaces/types.ts:47

MORPH\LOCAL\DEVNET

> MORPH\LOCAL\DEVNET: "http://localhost:8545"

Source

src/interfaces/types.ts:45

MORPH\TESTNET

> MORPH\TESTNET: "https://rpc-testnet.morphl2.io"

Source

src/interfaces/types.ts:46

MessageDirection.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / MessageDirection

Enumeration: MessageDirection

Enum describing the direction of a message.

Enumeration Members

L1\TO\L2

> L1\TO\L2: 0

Source

src/interfaces/types.ts:223

L2\TO\L1

> L2\TO\L1: 1

Source

src/interfaces/types.ts:224

MessageReceiptStatus.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / MessageReceiptStatus

Enumeration: MessageReceiptStatus

Enum describing the status of a CrossDomainMessage message receipt.

Enumeration Members

RELAYED\FAILED

> RELAYED\FAILED: 0

Source

src/interfaces/types.ts:299

RELAYED\SUCCEEDED

> RELAYED\SUCCEEDED: 1

Source

src/interfaces/types.ts:300

MessageStatus.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / MessageStatus

Enumeration: MessageStatus

Enum describing the status of a message.

Enumeration Members

FAILED\L1\TO\L2\MESSAGE

> FAILED\L1\TO\L2\MESSAGE: 1

Message is an L1 to L2 message and the transaction to execute the message failed.
When this status is returned, you will need to resend the L1 to L2 message, probably with a
higher gas limit.

Source

src/interfaces/types.ts:186

IN\CHALLENGE\PERIOD

> IN\CHALLENGE\PERIOD: 5

Message is a proved L2 to L1 message and is undergoing the challenge period.

Source

src/interfaces/types.ts:206

READY\FOR\RELAY

> READY\FOR\RELAY: 6

Message is ready to be relayed.

Source

src/interfaces/types.ts:211

READY\TO\PROVE

> READY\TO\PROVE: 4

Message is ready to be proved on L1 to initiate the challenge period.

Source

src/interfaces/types.ts:201

RELAYED

> RELAYED: 7

Message has been relayed.

Source

src/interfaces/types.ts:216

UNCONFIRMED\L1\TO\L2\MESSAGE

> UNCONFIRMED\L1\TO\L2\MESSAGE: 0

Message is an L1 to L2 message and has not been processed by the L2.

Source

src/interfaces/types.ts:179

WITHDRAWAL\HASH\NOT\SYNC

> WITHDRAWAL\HASH\NOT\SYNC: 3

Message is an L2 to L1 message and withdrawal hash has not been published to backend yet.

Source

src/interfaces/types.ts:196

WITHDRAWAL\ROOT\NOT\PUBLISHED

> WITHDRAWAL\ROOT\NOT\PUBLISHED: 2

Message is an L2 to L1 message and withdrawal root has not been published yet.

Source

src/interfaces/types.ts:191

asL2Provider.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / asL2Provider

Function: asL2Provider()

> asL2Provider\<TProvider\>(provider): L2Provider\<TProvider\>

Returns an provider wrapped as an Morph L2 provider. Adds a few extra helper functions to
simplify the process of estimating the gas usage for a transaction on Morph. Returns a COPY
of the original provider.

Type parameters

• TProvider extends Provider\<TProvider\>

Parameters

• provider: TProvider

Provider to wrap into an L2 provider.

Returns

L2Provider\<TProvider\>

Provider wrapped as an L2 provider.

Source

src/l2-provider.ts:171

estimateL1Gas.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / estimateL1Gas

Function: estimateL1Gas()

> estimateL1Gas(l2Provider, tx): Promise\<BigNumber\>

Estimates the amount of L1 gas required for a given L2 transaction.

Parameters

• l2Provider: ProviderLike

L2 provider to query the gas usage from.

• tx: TransactionRequest

Transaction to estimate L1 gas for.

Returns

Promise\<BigNumber\>

Estimated L1 gas.

Source

src/l2-provider.ts:71

estimateL1GasCost.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / estimateL1GasCost

Function: estimateL1GasCost()

> estimateL1GasCost(l2Provider, tx): Promise\<BigNumber\>

Estimates the amount of L1 gas cost for a given L2 transaction in wei.

Parameters

• l2Provider: ProviderLike

L2 provider to query the gas usage from.

• tx: TransactionRequest

Transaction to estimate L1 gas cost for.

Returns

Promise\<BigNumber\>

Estimated L1 gas cost.

Source

src/l2-provider.ts:95

estimateL2GasCost.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / estimateL2GasCost

Function: estimateL2GasCost()

> estimateL2GasCost(l2Provider, tx): Promise\<BigNumber\>

Estimates the L2 gas cost for a given L2 transaction in wei.

Parameters

• l2Provider: ProviderLike

L2 provider to query the gas usage from.

• tx: TransactionRequest

Transaction to estimate L2 gas cost for.

Returns

Promise\<BigNumber\>

Estimated L2 gas cost.

Source

src/l2-provider.ts:119

estimateTotalGasCost.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / estimateTotalGasCost

Function: estimateTotalGasCost()

> estimateTotalGasCost(l2Provider, tx): Promise\<BigNumber\>

Estimates the total gas cost for a given L2 transaction in wei.

Parameters

• l2Provider: ProviderLike

L2 provider to query the gas usage from.

• tx: TransactionRequest

Transaction to estimate total gas cost for.

Returns

Promise\<BigNumber\>

Estimated total gas cost.

Source

src/l2-provider.ts:136

getAllOEContracts.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / getAllOEContracts

Function: getAllOEContracts()

> getAllOEContracts(l2ChainId, opts): OEContracts

Automatically connects to all contract addresses, both L1 and L2, for the given L2 chain ID. The
user can provide custom contract address overrides for L1 or L2 contracts. If the given chain ID
is not known then the user MUST provide custom contract addresses for ALL L1 contracts or this
function will throw an error.

Parameters

• l2ChainId: number

Chain ID for the L2 network.

• opts= {}

Additional options for connecting to the contracts.

• opts.l1SignerOrProvider?: Provider \| Signer

• opts.l2SignerOrProvider?: Provider \| Signer

• opts.overrides?: DeepPartial\<OEContractsLike\>

Custom contract address overrides for L1 or L2 contracts.

Returns

OEContracts

An object containing ethers.Contract objects connected to the appropriate addresses on
both L1 and L2.

Source

src/utils/contracts.ts:88

getBridgeAdapters.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / getBridgeAdapters

Function: getBridgeAdapters()

> getBridgeAdapters(l2ChainId, messenger, opts?): BridgeAdapters

Gets a series of bridge adapters for the given L2 chain ID.

Parameters

• l2ChainId: number

Chain ID for the L2 network.

• messenger: CrossChainMessenger

Cross chain messenger to connect to the bridge adapters

• opts?

Additional options for connecting to the custom bridges.

• opts.contracts?: DeepPartial\<OEContractsLike\>

• opts.overrides?: BridgeAdapterData

Custom bridge adapters.

Returns

BridgeAdapters

An object containing all bridge adapters

Source

src/utils/contracts.ts:142

getL1GasPrice.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / getL1GasPrice

Function: getL1GasPrice()

> getL1GasPrice(l2Provider): Promise\<BigNumber\>

Gets the current L1 gas price as seen on L2.

Parameters

• l2Provider: ProviderLike

L2 provider to query the L1 gas price from.

Returns

Promise\<BigNumber\>

Current L1 gas price as seen on L2.

Source

src/l2-provider.ts:57

getOEContract.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / getOEContract

Function: getOEContract()

> getOEContract(contractName, l2ChainId, opts): Contract

Returns an ethers.Contract object for the given name, connected to the appropriate address for
the given L2 chain ID. Users can also provide a custom address to connect the contract to
instead. If the chain ID is not known then the user MUST provide a custom address or this
function will throw an error.

Parameters

• contractName: "L1MessageQueueWithGasPriceOracle" \| "L1GatewayRouter" \| "L2GatewayRouter" \| "MorphStandardERC20" \| "L2WETH" \| "L1WETHGateway" \| "L2WETHGateway" \| "L2ToL1MessagePasser" \| "Sequencer" \| "Gov" \| "L2ETHGateway" \| "L2CrossDomainMessenger" \| "L2StandardERC20Gateway" \| "L2ERC721Gateway" \| "L2TxFeeVault" \| "L2ERC1155Gateway" \| "MorphStandardERC20Factory" \| "GasPriceOracle" \| "WrappedEther" \| "MorphToken" \| "L1CrossDomainMessenger" \| "Staking" \| "Rollup" \| "L1ETHGateway" \| "L1StandardERC20Gateway" \| "L1ERC721Gateway" \| "L1ERC1155Gateway" \| "EnforcedTxGateway" \| "WETH"

Name of the contract to connect to.

• l2ChainId: number

Chain ID for the L2 network.

• opts= {}

Additional options for connecting to the contract.

• opts.address?: AddressLike

Custom address to connect to the contract.

• opts.signerOrProvider?: Provider \| Signer

Signer or provider to connect to the contract.

Returns

Contract

An ethers.Contract object connected to the appropriate address and interface.

Source

src/utils/contracts.ts:42

hashLowLevelMessageV2.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / hashLowLevelMessageV2

Function: hashLowLevelMessageV2()

> hashLowLevelMessageV2(message): string

Utility for hashing a LowLevelMessage object.

Parameters

• message: LowLevelMessage

LowLevelMessage object to hash.

Returns

string

Hash of the given LowLevelMessage.

Source

src/utils/message-utils.ts:82

hashMessageHash.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / hashMessageHash

Function: hashMessageHash()

> hashMessageHash(messageHash): string

Utility for hashing a message hash. This computes the storage slot where the message hash will be stored in state. HashZero is used because the first mapping in the contract is used.

Parameters

• messageHash: string

Message hash to hash.

Returns

string

Hash of the given message hash.

Source

src/utils/message-utils.ts:24

isL2Provider.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / isL2Provider

Function: isL2Provider()

> isL2Provider\<TProvider\>(provider): provider is L2Provider<TProvider>

Determines if a given Provider is an L2Provider. Will coerce type
if true

Type parameters

• TProvider extends Provider\<TProvider\>

Parameters

• provider: TProvider

The provider to check

Returns

provider is L2Provider<TProvider>

Boolean

Example

ts
if (isL2Provider(provider)) {
  // typescript now knows it is of type L2Provider
  const gasPrice = await provider.estimateL2GasPrice(tx)
}

Source

src/l2-provider.ts:157

migratedWithdrawalGasLimit.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / migratedWithdrawalGasLimit

Function: migratedWithdrawalGasLimit()

> migratedWithdrawalGasLimit(data, chainID): BigNumber

Compute the min gas limit for a migrated withdrawal.

Parameters

• data: string

• chainID: number

Returns

BigNumber

Source

src/utils/message-utils.ts:35

omit.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / omit

Function: omit()

> omit\<T, K\>(obj, ...keys): Omit\<T, K\>

Returns a copy of the given object (...obj) with the given keys omitted.

Type parameters

• T extends object

• K extends string \| number \| symbol

Parameters

• obj: T

Object to return with the keys omitted.

• ...keys: K[]

Keys to omit from the returned object.

Returns

Omit\<T, K\>

A copy of the given object with the given keys omitted.

Source

src/utils/misc-utils.ts:11

toAddress.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / toAddress

Function: toAddress()

> toAddress(addr): string

Converts an address-like into a 0x-prefixed address string.

Parameters

• addr: AddressLike

Address-like to convert into an address.

Returns

string

Address-like as an address.

Source

src/utils/coercion.ts:104

toBigNumber.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / toBigNumber

Function: toBigNumber()

> toBigNumber(num): BigNumber

Converts a number-like into an ethers BigNumber.

Parameters

• num: NumberLike

Number-like to convert into a BigNumber.

Returns

BigNumber

Number-like as a BigNumber.

Source

src/utils/coercion.ts:84

toNumber.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / toNumber

Function: toNumber()

> toNumber(num): number

Converts a number-like into a number.

Parameters

• num: NumberLike

Number-like to convert into a number.

Returns

number

Number-like as a number.

Source

src/utils/coercion.ts:94

toProvider.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / toProvider

Function: toProvider()

> toProvider(provider): Provider

Converts a ProviderLike into a Provider. Assumes that if the input is a string then it is a
JSON-RPC url.

Parameters

• provider: ProviderLike

ProviderLike to turn into a Provider.

Returns

Provider

Input as a Provider.

Source

src/utils/coercion.ts:46

toSignerOrProvider.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / toSignerOrProvider

Function: toSignerOrProvider()

> toSignerOrProvider(signerOrProvider): Provider \| Signer

Converts a SignerOrProviderLike into a Signer or a Provider. Assumes that if the input is a
string then it is a JSON-RPC url.

Parameters

• signerOrProvider: SignerOrProviderLike

SignerOrProviderLike to turn into a Signer or Provider.

Returns

Provider \| Signer

Input as a Signer or Provider.

Source

src/utils/coercion.ts:25

toTransactionHash.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / toTransactionHash

Function: toTransactionHash()

> toTransactionHash(transaction): string

Pulls a transaction hash out of a TransactionLike object.

Parameters

• transaction: TransactionLike

TransactionLike to convert into a transaction hash.

Returns

string

Transaction hash corresponding to the TransactionLike input.

Source

src/utils/coercion.ts:62

BridgeAdapterData.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / BridgeAdapterData

Interface: BridgeAdapterData

Something that looks like the list of custom bridges.

Indexable

\[name: string\]: object

BridgeAdapters.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / BridgeAdapters

Interface: BridgeAdapters

Something that looks like the list of custom bridges.

Indexable

 \[name: string\]: IBridgeAdapter

CoreCrossChainMessage.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / CoreCrossChainMessage

Interface: CoreCrossChainMessage

Core components of a cross chain message.

Extended by

- CrossChainMessage

Properties

message

> message: string

Source

src/interfaces/types.ts:242

messageNonce

> messageNonce: BigNumber

Source

src/interfaces/types.ts:243

minGasLimit

> minGasLimit: BigNumber

Source

src/interfaces/types.ts:245

sender

> sender: string

Source

src/interfaces/types.ts:240

target

> target: string

Source

src/interfaces/types.ts:241

value

> value: BigNumber

Source

src/interfaces/types.ts:244

CrossChainMessage.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / CrossChainMessage

Interface: CrossChainMessage

Describes a message that is sent between L1 and L2. Direction determines where the message was
sent from and where it's being sent to.

Extends

- CoreCrossChainMessage

Properties

blockNumber

> blockNumber: number

Source

src/interfaces/types.ts:255

direction

> direction: MessageDirection

Source

src/interfaces/types.ts:253

logIndex

> logIndex: number

Source

src/interfaces/types.ts:254

message

> message: string

Inherited from

CoreCrossChainMessage.message

Source

src/interfaces/types.ts:242

messageNonce

> messageNonce: BigNumber

Inherited from

CoreCrossChainMessage.messageNonce

Source

src/interfaces/types.ts:243

minGasLimit

> minGasLimit: BigNumber

Inherited from

CoreCrossChainMessage.minGasLimit

Source

src/interfaces/types.ts:245

sender

> sender: string

Inherited from

CoreCrossChainMessage.sender

Source

src/interfaces/types.ts:240

target

> target: string

Inherited from

CoreCrossChainMessage.target

Source

src/interfaces/types.ts:241

transactionHash

> transactionHash: string

Source

src/interfaces/types.ts:256

value

> value: BigNumber

Inherited from

CoreCrossChainMessage.value

Source

src/interfaces/types.ts:244

CrossChainMessageRequest.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / CrossChainMessageRequest

Interface: CrossChainMessageRequest

Partial message that needs to be signed and executed by a specific signer.

Properties

direction

> direction: MessageDirection

Source

src/interfaces/types.ts:231

message

> message: string

Source

src/interfaces/types.ts:233

target

> target: string

Source

src/interfaces/types.ts:232

IActionOptions.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / IActionOptions

Interface: IActionOptions

Param

Optional signer to use to send the transaction.

Param

Optional address to receive the funds on chain. Defaults to sender.

Param

Direction to search for messages in. If not provided, will attempt to
   automatically search both directions under the assumption that a transaction
hash will only
   exist on one chain. If the hash exists on both chains, will throw an error.

Param

Optional transaction overrides.

Properties

direction?

> optional direction: MessageDirection

Source

src/interfaces/types.ts:412

from?

> optional from: string

Source

src/interfaces/types.ts:410

overrides?

> optional overrides: object & CallOverrides

Type declaration

gatewayAddress?

> optional gatewayAddress: string

gatewayName?

> optional gatewayName: string

Source

src/interfaces/types.ts:413

recipient?

> optional recipient: AddressLike

Source

src/interfaces/types.ts:411

signer?

> optional signer: Signer

Source

src/interfaces/types.ts:409

IBridgeAdapter.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / IBridgeAdapter

Interface: IBridgeAdapter

Represents an adapter for an L1 - L2 token bridge. Each custom bridge currently needs its own
adapter because the bridge interface is not standardized. This may change in the future.

Properties

estimateGas

> estimateGas: object

Object that holds the functions that estimates the gas required for a given transaction.
Follows the pattern used by ethers.js.

approve()

Estimates gas required to approve some tokens to deposit into the L2 chain.

Parameters

• l1Token: AddressLike

The L1 token address.

• l2Token: AddressLike

The L2 token address.

• amount: NumberLike

Amount of the token to approve.

• opts?: IActionOptions

Additional options.

Returns

Promise\<BigNumber\>

Gas estimate for the transaction.

deposit()

Estimates gas required to deposit some tokens into the L2 chain.

Parameters

• l1Token: AddressLike

The L1 token address.

• l2Token: AddressLike

The L2 token address.

• amount: NumberLike

Amount of the token to deposit.

• opts?: IActionOptions

Additional options.

Returns

Promise\<BigNumber\>

Gas estimate for the transaction.

withdraw()

Estimates gas required to withdraw some tokens back to the L1 chain.

Parameters

• l1Token: AddressLike

The L1 token address.

• l2Token: AddressLike

The L2 token address.

• amount: NumberLike

Amount of the token to withdraw.

• opts?: IActionOptions

Additional options.

Returns

Promise\<BigNumber\>

Gas estimate for the transaction.

Source

src/interfaces/bridge-adapter.ts:221

l1Bridge

> l1Bridge: Contract

L1 bridge contract.

Source

src/interfaces/bridge-adapter.ts:38

l2Bridge

> l2Bridge: Contract

L2 bridge contract.

Source

src/interfaces/bridge-adapter.ts:43

messenger

> messenger: CrossChainMessenger

Provider used to make queries related to cross-chain interactions.

Source

src/interfaces/bridge-adapter.ts:33

populateTransaction

> populateTransaction: object

Object that holds the functions that generate transactions to be signed by the user.
Follows the pattern used by ethers.js.

approve()

Generates a transaction for approving some tokens to deposit into the L2 chain.

Parameters

• l1Token: AddressLike

The L1 token address.

• l2Token: AddressLike

The L2 token address.

• amount: NumberLike

Amount of the token to approve.

• opts?: IActionOptions

Additional options.

Returns

Promise\<TransactionRequest\>

Transaction that can be signed and executed to deposit the tokens.

deposit()

Generates a transaction for depositing some tokens into the L2 chain.

Parameters

• l1Token: AddressLike

The L1 token address.

• l2Token: AddressLike

The L2 token address.

• amount: NumberLike

Amount of the token to deposit.

• opts?: IActionOptions

Additional options.

Returns

Promise\<TransactionRequest\>

Transaction that can be signed and executed to deposit the tokens.

withdraw()

Generates a transaction for withdrawing some tokens back to the L1 chain.

Parameters

• l1Token: AddressLike

The L1 token address.

• l2Token: AddressLike

The L2 token address.

• amount: NumberLike

Amount of the token to withdraw.

• opts?: IActionOptions

Additional options.

Returns

Promise\<TransactionRequest\>

Transaction that can be signed and executed to withdraw the tokens.

Source

src/interfaces/bridge-adapter.ts:167

Methods

approval()

> approval(l1Token, l2Token, opts?): Promise\<BigNumber\>

Queries the account's approval amount for a given L1 token.

Parameters

• l1Token: AddressLike

The L1 token address.

• l2Token: AddressLike

The L2 token address.

• opts?: IActionOptions

Additional options.

Returns

Promise\<BigNumber\>

Amount of tokens approved for deposits from the account.

Source

src/interfaces/bridge-adapter.ts:103

approve()

> approve(l1Token, l2Token, amount, signer, opts?): Promise\
<TransactionResponse\>

Approves a deposit into the L2 chain.

Parameters

• l1Token: AddressLike

The L1 token address.

• l2Token: AddressLike

The L2 token address.

• amount: NumberLike

Amount of the token to approve.

• signer: Signer

Signer used to sign and send the transaction.

• opts?: IActionOptions

Additional options.

Returns

Promise\<TransactionResponse\>

Transaction response for the approval transaction.

Source

src/interfaces/bridge-adapter.ts:119

deposit()

> deposit(l1Token, l2Token, amount, signer, opts?): Promise\
<TransactionResponse\>

Deposits some tokens into the L2 chain.

Parameters

• l1Token: AddressLike

The L1 token address.

• l2Token: AddressLike

The L2 token address.

• amount: NumberLike

Amount of the token to deposit.

• signer: Signer

Signer used to sign and send the transaction.

• opts?: IActionOptions

Additional options.

Returns

Promise\<TransactionResponse\>

Transaction response for the deposit transaction.

Source

src/interfaces/bridge-adapter.ts:137

getDepositsByAddress()

> getDepositsByAddress(address, opts?): Promise\<TokenBridgeMessage[]\>

Gets all deposits for a given address.

Parameters

• address: AddressLike

Address to search for messages from.

• opts?

Options object.

• opts.fromBlock?: BlockTag

Block to start searching for messages from. If not provided, will start from the first block (block #0).

• opts.toBlock?: BlockTag

Block to stop searching for messages at. If not provided, will stop at the latest known block ("latest").

Returns

Promise\<TokenBridgeMessage[]\>

All deposit token bridge messages sent by the given address.

Source

src/interfaces/bridge-adapter.ts:56

getWithdrawalsByAddress()

> getWithdrawalsByAddress(address, opts?): Promise\<TokenBridgeMessage[]\>

Gets all withdrawals for a given address.

Parameters

• address: AddressLike

Address to search for messages from.

• opts?

Options object.

• opts.fromBlock?: BlockTag

Block to start searching for messages from. If not provided, will start from the first block (block #0).

• opts.toBlock?: BlockTag

Block to stop searching for messages at. If not provided, will stop at the latest known block ("latest").

Returns

Promise\<TokenBridgeMessage[]\>

All withdrawal token bridge messages sent by the given address.

Source

src/interfaces/bridge-adapter.ts:75

supportsTokenPair()

> supportsTokenPair(l1Token, l2Token): Promise\<boolean\>

Checks whether the given token pair is supported by the bridge.

Parameters

• l1Token: AddressLike

The L1 token address.

• l2Token: AddressLike

The L2 token address.

Returns

Promise\<boolean\>

Whether the given token pair is supported by the bridge.

Source

src/interfaces/bridge-adapter.ts:90

withdraw()

> withdraw(l1Token, l2Token, amount, signer, opts?): Promise\
<TransactionResponse\>

Withdraws some tokens back to the L1 chain.

Parameters

• l1Token: AddressLike

The L1 token address.

• l2Token: AddressLike

The L2 token address.

• amount: NumberLike

Amount of the token to withdraw.

• signer: Signer

Signer used to sign and send the transaction.

• opts?: IActionOptions

Additional options.

Returns

Promise\<TransactionResponse\>

Transaction response for the withdraw transaction.

Source

src/interfaces/bridge-adapter.ts:155

L2Block.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / L2Block

Interface: L2Block

JSON block representation when returned by L2Geth nodes. Just a normal block but with
an added stateRoot field.

Extends

- Block

Properties

\difficulty

> \difficulty: BigNumber

Inherited from

Block.difficulty

Source

node\modules/@ethersproject/abstract-provider/lib/index.d.ts:41

baseFeePerGas?

> optional baseFeePerGas: BigNumber

Inherited from

Block.baseFeePerGas

Source

node\modules/@ethersproject/abstract-provider/lib/index.d.ts:46

difficulty

> difficulty: number

Inherited from

Block.difficulty

Source

node\modules/@ethersproject/abstract-provider/lib/index.d.ts:40

extraData

> extraData: string

Inherited from

Block.extraData

Source

node\modules/@ethersproject/abstract-provider/lib/index.d.ts:45

gasLimit

> gasLimit: BigNumber

Inherited from

Block.gasLimit

Source

node\modules/@ethersproject/abstract-provider/lib/index.d.ts:42

gasUsed

> gasUsed: BigNumber

Inherited from

Block.gasUsed

Source

node\modules/@ethersproject/abstract-provider/lib/index.d.ts:43

hash

> hash: string

Inherited from

Block.hash

Source

node\modules/@ethersproject/abstract-provider/lib/index.d.ts:35

miner

> miner: string

Inherited from

Block.miner

Source

node\modules/@ethersproject/abstract-provider/lib/index.d.ts:44

nonce

> nonce: string

Inherited from

Block.nonce

Source

node\modules/@ethersproject/abstract-provider/lib/index.d.ts:39

number

> number: number

Inherited from

Block.number

Source

node\modules/@ethersproject/abstract-provider/lib/index.d.ts:37

parentHash

> parentHash: string

Inherited from

Block.parentHash

Source

node\modules/@ethersproject/abstract-provider/lib/index.d.ts:36

stateRoot

> stateRoot: string

Source

src/interfaces/l2-provider.ts:27

timestamp

> timestamp: number

Inherited from

Block.timestamp

Source

node\modules/@ethersproject/abstract-provider/lib/index.d.ts:38

transactions

> transactions: string[]

Inherited from

Block.transactions

Source

node\modules/@ethersproject/abstract-provider/lib/index.d.ts:49

L2BlockWithTransactions.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / L2BlockWithTransactions

Interface: L2BlockWithTransactions

JSON block representation when returned by L2Geth nodes. Just a normal block but with
L2Transaction objects instead of the standard transaction response object.

Extends

- BlockWithTransactions

Properties

\difficulty

> \difficulty: BigNumber

Inherited from

BlockWithTransactions.difficulty

Source

node\modules/@ethersproject/abstract-provider/lib/index.d.ts:41

baseFeePerGas?

> optional baseFeePerGas: BigNumber

Inherited from

BlockWithTransactions.baseFeePerGas

Source

node\modules/@ethersproject/abstract-provider/lib/index.d.ts:46

difficulty

> difficulty: number

Inherited from

BlockWithTransactions.difficulty

Source

node\modules/@ethersproject/abstract-provider/lib/index.d.ts:40

extraData

> extraData: string

Inherited from

BlockWithTransactions.extraData

Source

node\modules/@ethersproject/abstract-provider/lib/index.d.ts:45

gasLimit

> gasLimit: BigNumber

Inherited from

BlockWithTransactions.gasLimit

Source

node\modules/@ethersproject/abstract-provider/lib/index.d.ts:42

gasUsed

> gasUsed: BigNumber

Inherited from

BlockWithTransactions.gasUsed

Source

node\modules/@ethersproject/abstract-provider/lib/index.d.ts:43

hash

> hash: string

Inherited from

BlockWithTransactions.hash

Source

node\modules/@ethersproject/abstract-provider/lib/index.d.ts:35

miner

> miner: string

Inherited from

BlockWithTransactions.miner

Source

node\modules/@ethersproject/abstract-provider/lib/index.d.ts:44

nonce

> nonce: string

Inherited from

BlockWithTransactions.nonce

Source

node\modules/@ethersproject/abstract-provider/lib/index.d.ts:39

number

> number: number

Inherited from

BlockWithTransactions.number

Source

node\modules/@ethersproject/abstract-provider/lib/index.d.ts:37

parentHash

> parentHash: string

Inherited from

BlockWithTransactions.parentHash

Source

node\modules/@ethersproject/abstract-provider/lib/index.d.ts:36

stateRoot

> stateRoot: string

Source

src/interfaces/l2-provider.ts:35

timestamp

> timestamp: number

Inherited from

BlockWithTransactions.timestamp

Source

node\modules/@ethersproject/abstract-provider/lib/index.d.ts:38

transactions

> transactions: [L2Transaction]

Overrides

BlockWithTransactions.transactions

Source

src/interfaces/l2-provider.ts:36

L2Transaction.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / L2Transaction

Interface: L2Transaction

JSON transaction representation when returned by L2Geth nodes. This is simply an extension to
the standard transaction response type. You do NOT need to use this type unless you care about
having typed access to L2-specific fields.

Extends

- TransactionResponse

Properties

accessList?

> optional accessList: AccessList

Inherited from

TransactionResponse.accessList

Source

node\modules/@ethersproject/transactions/lib/index.d.ts:40

blockHash?

> optional blockHash: string

Inherited from

TransactionResponse.blockHash

Source

node\modules/@ethersproject/abstract-provider/lib/index.d.ts:26

blockNumber?

> optional blockNumber: number

Inherited from

TransactionResponse.blockNumber

Source

node\modules/@ethersproject/abstract-provider/lib/index.d.ts:25

chainId

> chainId: number

Inherited from

TransactionResponse.chainId

Source

node\modules/@ethersproject/transactions/lib/index.d.ts:35

confirmations

> confirmations: number

Inherited from

TransactionResponse.confirmations

Source

node\modules/@ethersproject/abstract-provider/lib/index.d.ts:28

data

> data: string

Inherited from

TransactionResponse.data

Source

node\modules/@ethersproject/transactions/lib/index.d.ts:33

from

> from: string

Inherited from

TransactionResponse.from

Source

node\modules/@ethersproject/abstract-provider/lib/index.d.ts:29

gasLimit

> gasLimit: BigNumber

Inherited from

TransactionResponse.gasLimit

Source

node\modules/@ethersproject/transactions/lib/index.d.ts:31

gasPrice?

> optional gasPrice: BigNumber

Inherited from

TransactionResponse.gasPrice

Source

node\modules/@ethersproject/transactions/lib/index.d.ts:32

hash

> hash: string

Inherited from

TransactionResponse.hash

Source

node\modules/@ethersproject/abstract-provider/lib/index.d.ts:24

l1BlockNumber

> l1BlockNumber: number

Source

src/interfaces/l2-provider.ts:16

l1TxOrigin

> l1TxOrigin: string

Source

src/interfaces/l2-provider.ts:17

maxFeePerGas?

> optional maxFeePerGas: BigNumber

Inherited from

TransactionResponse.maxFeePerGas

Source

node\modules/@ethersproject/transactions/lib/index.d.ts:42

maxPriorityFeePerGas?

> optional maxPriorityFeePerGas: BigNumber

Inherited from

TransactionResponse.maxPriorityFeePerGas

Source

node\modules/@ethersproject/transactions/lib/index.d.ts:41

nonce

> nonce: number

Inherited from

TransactionResponse.nonce

Source

node\modules/@ethersproject/transactions/lib/index.d.ts:30

queueOrigin

> queueOrigin: string

Source

src/interfaces/l2-provider.ts:18

r?

> optional r: string

Inherited from

TransactionResponse.r

Source

node\modules/@ethersproject/transactions/lib/index.d.ts:36

raw?

> optional raw: string

Inherited from

TransactionResponse.raw

Source

node\modules/@ethersproject/abstract-provider/lib/index.d.ts:30

rawTransaction

> rawTransaction: string

Source

src/interfaces/l2-provider.ts:19

s?

> optional s: string

Inherited from

TransactionResponse.s

Source

node\modules/@ethersproject/transactions/lib/index.d.ts:37

timestamp?

> optional timestamp: number

Inherited from

TransactionResponse.timestamp

Source

node\modules/@ethersproject/abstract-provider/lib/index.d.ts:27

to?

> optional to: string

Inherited from

TransactionResponse.to

Source

node\modules/@ethersproject/transactions/lib/index.d.ts:28

type?

> optional type: number

Inherited from

TransactionResponse.type

Source

node\modules/@ethersproject/transactions/lib/index.d.ts:39

v?

> optional v: number

Inherited from

TransactionResponse.v

Source

node\modules/@ethersproject/transactions/lib/index.d.ts:38

value

> value: BigNumber

Inherited from

TransactionResponse.value

Source

node\modules/@ethersproject/transactions/lib/index.d.ts:34

wait()

> wait: (confirmations?) => Promise\<TransactionReceipt\>

Parameters

• confirmations?: number

Returns

Promise\<TransactionReceipt\>

Inherited from

TransactionResponse.wait

Source

node\modules/@ethersproject/abstract-provider/lib/index.d.ts:31

MessageReceipt.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / MessageReceipt

Interface: MessageReceipt

CrossDomainMessage receipt.

Properties

receiptStatus

> receiptStatus: MessageReceiptStatus

Source

src/interfaces/types.ts:307

transactionReceipt?

> optional transactionReceipt: TransactionReceipt

Source

src/interfaces/types.ts:308

OEContracts.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / OEContracts

Interface: OEContracts

Represents Morph contracts, assumed to be connected to their appropriate providers and addresses.

Properties

l1

> l1: OEL1Contracts

Source

src/interfaces/types.ts:121

l2

> l2: OEL2Contracts

Source

src/interfaces/types.ts:122

OEContractsLike.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / OEContractsLike

Interface: OEContractsLike

Convenience type for something that looks like the OE contract interface but could be
addresses instead of actual contract objects.

Properties

l1

> l1: OEL1ContractsLike

Source

src/interfaces/types.ts:146

l2

> l2: OEL2ContractsLike

Source

src/interfaces/types.ts:147

OEL1Contracts.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / OEL1Contracts

Interface: OEL1Contracts

L1 contract references.

Properties

EnforcedTxGateway

> EnforcedTxGateway: Contract

Source

src/interfaces/types.ts:64

L1CrossDomainMessenger

> L1CrossDomainMessenger: Contract

Source

src/interfaces/types.ts:54

L1ERC1155Gateway

> L1ERC1155Gateway: Contract

Source

src/interfaces/types.ts:63

L1ERC721Gateway

> L1ERC721Gateway: Contract

Source

src/interfaces/types.ts:62

L1ETHGateway

> L1ETHGateway: Contract

Source

src/interfaces/types.ts:60

L1GatewayRouter

> L1GatewayRouter: Contract

Source

src/interfaces/types.ts:59

L1MessageQueueWithGasPriceOracle

> L1MessageQueueWithGasPriceOracle: Contract

Source

src/interfaces/types.ts:55

L1StandardERC20Gateway

> L1StandardERC20Gateway: Contract

Source

src/interfaces/types.ts:61

L1WETHGateway

> L1WETHGateway: Contract

Source

src/interfaces/types.ts:66

MorphToken?

> optional MorphToken: Contract

Source

src/interfaces/types.ts:68

Rollup

> Rollup: Contract

Source

src/interfaces/types.ts:58

Staking

> Staking: Contract

Source

src/interfaces/types.ts:56

WETH

> WETH: Contract

Source

src/interfaces/types.ts:65

OEL2Contracts.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / OEL2Contracts

Interface: OEL2Contracts

L2 contract references.

Properties

GasPriceOracle

> GasPriceOracle: Contract

Source

src/interfaces/types.ts:97

Gov

> Gov: Contract

Source

src/interfaces/types.ts:86

L2CrossDomainMessenger

> L2CrossDomainMessenger: Contract

Source

src/interfaces/types.ts:89

L2ERC1155Gateway

> L2ERC1155Gateway: Contract

Source

src/interfaces/types.ts:94

L2ERC721Gateway

> L2ERC721Gateway: Contract

Source

src/interfaces/types.ts:91

L2ETHGateway

> L2ETHGateway: Contract

Source

src/interfaces/types.ts:88

L2GatewayRouter

> L2GatewayRouter: Contract

Source

src/interfaces/types.ts:84

L2StandardERC20Gateway

> L2StandardERC20Gateway: Contract

Source

src/interfaces/types.ts:90

L2ToL1MessagePasser

> L2ToL1MessagePasser: Contract

Source

src/interfaces/types.ts:83

L2TxFeeVault

> L2TxFeeVault: Contract

Source

src/interfaces/types.ts:92

L2WETH

> L2WETH: Contract

Source

src/interfaces/types.ts:98

L2WETHGateway

> L2WETHGateway: Contract

Source

src/interfaces/types.ts:99

MorphStandardERC20

> MorphStandardERC20: Contract

Source

src/interfaces/types.ts:95

MorphStandardERC20Factory

> MorphStandardERC20Factory: Contract

Source

src/interfaces/types.ts:96

MorphToken?

> optional MorphToken: Contract

Source

src/interfaces/types.ts:102

Sequencer

> Sequencer: Contract

Source

src/interfaces/types.ts:85

WrappedEther

> WrappedEther: Contract

Source

src/interfaces/types.ts:100

ProvenWithdrawal.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / ProvenWithdrawal

Interface: ProvenWithdrawal

ProvenWithdrawal in L1CrossDomainMessenger

Properties

timestamp

> timestamp: BigNumber

Source

src/interfaces/types.ts:316

withdrawalIndex

> withdrawalIndex: BigNumber

Source

src/interfaces/types.ts:319

withdrawalRoot

> withdrawalRoot: string

Source

src/interfaces/types.ts:318

StateRoot.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / StateRoot

Interface: StateRoot

Information about a state root, including header, block number, and root iself.

Properties

batch

> batch: StateRootBatch

Source

src/interfaces/types.ts:339

stateRoot

> stateRoot: string

Source

src/interfaces/types.ts:337

stateRootIndexInBatch

> stateRootIndexInBatch: number

Source

src/interfaces/types.ts:338

StateRootBatch.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / StateRootBatch

Interface: StateRootBatch

Information about a batch of state roots.

Properties

blockNumber

> blockNumber: number

Source

src/interfaces/types.ts:346

header

> header: StateRootBatchHeader

Source

src/interfaces/types.ts:347

stateRoots

> stateRoots: string[]

Source

src/interfaces/types.ts:348

StateRootBatchHeader.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / StateRootBatchHeader

Interface: StateRootBatchHeader

Header for a state root batch.

Properties

batchIndex

> batchIndex: BigNumber

Source

src/interfaces/types.ts:326

batchRoot

> batchRoot: string

Source

src/interfaces/types.ts:327

batchSize

> batchSize: BigNumber

Source

src/interfaces/types.ts:328

extraData

> extraData: string

Source

src/interfaces/types.ts:330

prevTotalElements

> prevTotalElements: BigNumber

Source

src/interfaces/types.ts:329

TokenBridgeMessage.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / TokenBridgeMessage

Interface: TokenBridgeMessage

Describes a token withdrawal or deposit, along with the underlying raw cross chain message
behind the deposit or withdrawal.

Properties

amount

> amount: BigNumber

Source

src/interfaces/types.ts:280

blockNumber

> blockNumber: number

Source

src/interfaces/types.ts:283

data

> data: string

Source

src/interfaces/types.ts:281

direction

> direction: MessageDirection

Source

src/interfaces/types.ts:275

from

> from: string

Source

src/interfaces/types.ts:276

l1Token

> l1Token: string

Source

src/interfaces/types.ts:278

l2Token

> l2Token: string

Source

src/interfaces/types.ts:279

logIndex

> logIndex: number

Source

src/interfaces/types.ts:282

to

> to: string

Source

src/interfaces/types.ts:277

transactionHash

> transactionHash: string

Source

src/interfaces/types.ts:284

WithdrawalEntry.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / WithdrawalEntry

Interface: WithdrawalEntry

Represents a withdrawal entry within the logs of a L2 to L1 CrossChainMessage

Properties

MessagePassed

> MessagePassed: any

Source

src/interfaces/types.ts:292

WithdrawMessageProof.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / WithdrawMessageProof

Interface: WithdrawMessageProof

Properties

withdrawalIndex

> withdrawalIndex: BigNumber

Source

src/cross-chain-messenger.ts:61

withdrawalLeaf

> withdrawalLeaf: any

Source

src/cross-chain-messenger.ts:64

withdrawalProof

> withdrawalProof: string[]

Source

src/cross-chain-messenger.ts:62

withdrawalRoot

> withdrawalRoot: string

Source

src/cross-chain-messenger.ts:63

AddressLike.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / AddressLike

Type alias: AddressLike

> AddressLike: string \| Contract

Stuff that can be coerced into an address.

Source

src/interfaces/types.ts:391

DeepPartial.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / DeepPartial

Type alias: DeepPartial\<T\>

> DeepPartial\<T\>: { [P in keyof T]?: DeepPartial<T[P]> }

Utility type for deep partials.

Type parameters

• T

Source

src/utils/type-utils.ts:4

L1Provider.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / L1Provider

Type alias: L1Provider\<TProvider\>

> L1Provider\<TProvider\>: TProvider & object

Represents an extended version of an normal ethers Provider that returns additional L1 info and
has special functions for L1-specific interactions.

Type declaration

\isL1Provider

> \isL1Provider: true

Internal property to determine if a provider is a L1Provider
You are likely looking for the isL2Provider function

estimateCrossDomainMessageFee()

Gets the current L1 (data) gas price.

Parameters

• l1Provider: ProviderLike

• sender: string

• gasLimit: number \| bigint \| BigNumber

Returns

Promise\<BigNumber\>

Current L1 data gas price in wei.

Type parameters

• TProvider extends Provider

Source

src/interfaces/l1-provider.ts:11

L2Provider.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / L2Provider

Type alias: L2Provider\<TProvider\>

> L2Provider\<TProvider\>: TProvider & object

Represents an extended version of an normal ethers Provider that returns additional L2 info and
has special functions for L2-specific interactions.

Type declaration

\isL2Provider

> \isL2Provider: true

Internal property to determine if a provider is a L2Provider
You are likely looking for the isL2Provider function

estimateL1Gas()

Estimates the L1 (data) gas required for a transaction.

Parameters

• tx: TransactionRequest

Transaction to estimate L1 gas for.

Returns

Promise\<BigNumber\>

Estimated L1 gas.

estimateL1GasCost()

Estimates the L1 (data) gas cost for a transaction in wei by multiplying the estimated L1 gas
cost by the current L1 gas price.

Parameters

• tx: TransactionRequest

Transaction to estimate L1 gas cost for.

Returns

Promise\<BigNumber\>

Estimated L1 gas cost.

estimateL2GasCost()

Estimates the L2 (execution) gas cost for a transaction in wei by multiplying the estimated L1
gas cost by the current L2 gas price. This is a simple multiplication of the result of
getGasPrice and estimateGas for the given transaction request.

Parameters

• tx: TransactionRequest

Transaction to estimate L2 gas cost for.

Returns

Promise\<BigNumber\>

Estimated L2 gas cost.

estimateTotalGasCost()

Estimates the total gas cost for a transaction in wei by adding the estimated the L1 gas cost
and the estimated L2 gas cost.

Parameters

• tx: TransactionRequest

Transaction to estimate total gas cost for.

Returns

Promise\<BigNumber\>

Estimated total gas cost.

getL1GasPrice()

Gets the current L1 (data) gas price.

Returns

Promise\<BigNumber\>

Current L1 data gas price in wei.

Type parameters

• TProvider extends Provider

Source

src/interfaces/l2-provider.ts:43

LowLevelMessage.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / LowLevelMessage

Type alias: LowLevelMessage

> LowLevelMessage: CoreCrossChainMessage & object

Describes messages sent inside the L2ToL1MessagePasser on L2. Happens to be the same structure
as the CoreCrossChainMessage so we'll reuse the type for now.

Type declaration

encodedMessage

> encodedMessage: string

messageHash

> messageHash: string

messageSender

> messageSender: string

messageTarget

> messageTarget: string

Source

src/interfaces/types.ts:263

MessageLike.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / MessageLike

Type alias: MessageLike

> MessageLike: CrossChainMessage \| TransactionLike \| TokenBridgeMessage

Stuff that can be coerced into a CrossChainMessage.

Source

src/interfaces/types.ts:359

MessageRequestLike.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / MessageRequestLike

Type alias: MessageRequestLike

> MessageRequestLike: CrossChainMessageRequest \| CrossChainMessage \|
TransactionLike \| TokenBridgeMessage

Stuff that can be coerced into a CrossChainMessageRequest.

Source

src/interfaces/types.ts:367

NumberLike.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / NumberLike

Type alias: NumberLike

> NumberLike: string \| number \| BigNumber \| bigint

Stuff that can be coerced into a number.

Source

src/interfaces/types.ts:396

OEL1ContractsLike.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / OEL1ContractsLike

Type alias: OEL1ContractsLike

> OEL1ContractsLike: { [K in keyof OEL1Contracts]: AddressLike }

Convenience type for something that looks like the L1 OE contract interface but
could be
addresses instead of actual contract objects.

Source

src/interfaces/types.ts:129

OEL2ContractsLike.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / OEL2ContractsLike

Type alias: OEL2ContractsLike

> OEL2ContractsLike: { [K in keyof OEL2Contracts]: AddressLike }

Convenience type for something that looks like the L2 OE contract interface but could be
addresses instead of actual contract objects.

Source

src/interfaces/types.ts:137

ProviderLike.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / ProviderLike

Type alias: ProviderLike

> ProviderLike: string \| Provider

Stuff that can be coerced into a provider.

Source

src/interfaces/types.ts:376

SignerLike.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / SignerLike

Type alias: SignerLike

> SignerLike: string \| Signer

Stuff that can be coerced into a signer.

Source

src/interfaces/types.ts:381

SignerOrProviderLike.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / SignerOrProviderLike

Type alias: SignerOrProviderLike

> SignerOrProviderLike: SignerLike \| ProviderLike

Stuff that can be coerced into a signer or provider.

Source

src/interfaces/types.ts:386

TransactionLike.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / TransactionLike

Type alias: TransactionLike

> TransactionLike: string \| TransactionReceipt \| TransactionResponse

Stuff that can be coerced into a transaction.

Source

src/interfaces/types.ts:354

BRIDGE_ADAPTER_DATA.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / BRIDGE\ADAPTER\DATA

Variable: BRIDGE\ADAPTER\DATA

> const BRIDGE\ADAPTER\DATA: { [ChainID in L2ChainID]?: BridgeAdapterData }

Mapping of L1 chain IDs to the list of custom bridge addresses for each chain.

Source

src/utils/chain-constants.ts:128

CHAIN_BLOCK_TIMES.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / CHAIN\BLOCK\TIMES

Variable: CHAIN\BLOCK\TIMES

> const CHAIN\BLOCK\TIMES: { [ChainID in L1ChainID]: number }

Source

src/utils/chain-constants.ts:22

CONTRACT_ADDRESSES.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / CONTRACT\ADDRESSES

Variable: CONTRACT\ADDRESSES

> const CONTRACT\ADDRESSES: { [ChainID in L2ChainID]: OEContractsLike }

Mapping of L1 chain IDs to the appropriate contract addresses for the OE deployments to the
given network. Simplifies the process of getting the correct contract addresses for a given
contract name.

Source

src/utils/chain-constants.ts:100

DEFAULT_L1_CONTRACT_ADDRESSES.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / DEFAULT\L1\CONTRACT\ADDRESSES

Variable: DEFAULT\L1\CONTRACT\ADDRESSES

> const DEFAULT\L1\CONTRACT\ADDRESSES: OEL1ContractsLike

Full list of default L1 contract addresses.

Source

src/utils/chain-constants.ts:61

DEFAULT_L2_CONTRACT_ADDRESSES.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / DEFAULT\L2\CONTRACT\ADDRESSES

Variable: DEFAULT\L2\CONTRACT\ADDRESSES

> const DEFAULT\L2\CONTRACT\ADDRESSES: OEL2ContractsLike

Full list of default L2 contract addresses.

Source

src/utils/chain-constants.ts:37

DEPOSIT_CONFIRMATION_BLOCKS.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / DEPOSIT\CONFIRMATION\BLOCKS

Variable: DEPOSIT\CONFIRMATION\BLOCKS

> const DEPOSIT\CONFIRMATION\BLOCKS: { [ChainID in L2ChainID]: number }

Source

src/utils/chain-constants.ts:12

l1BridgeName.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / l1BridgeName

Variable: l1BridgeName

> const l1BridgeName: "L1GatewayRouter" = 'L1GatewayRouter'

Source

src/cross-chain-messenger.ts:69

l1CrossDomainMessengerName.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / l1CrossDomainMessengerName

Variable: l1CrossDomainMessengerName

> const l1CrossDomainMessengerName: "L1CrossDomainMessenger" =
'L1CrossDomainMessenger'

Source

src/cross-chain-messenger.ts:67

l2BridgeName.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / l2BridgeName

Variable: l2BridgeName

> const l2BridgeName: "L2GatewayRouter" = 'L2GatewayRouter'

Source

src/cross-chain-messenger.ts:70

l2CrossDomainMessengerName.md:

@morph-l2/sdk • Docs

@morph-l2/sdk / l2CrossDomainMessengerName

Variable: l2CrossDomainMessengerName

> const l2CrossDomainMessengerName: "L2CrossDomainMessenger" =
'L2CrossDomainMessenger'

Source

src/cross-chain-messenger.ts:68

1-intro.md:

---
title: Introduction
lang: en-US
keywords: [morph,ethereum,rollup,layer2,validity proof,optimistic zk-rollup]
description: Upgrade your blockchain experience with Morph - the secure decentralized, cost0efficient, and high-performing optimistic zk-rollup solution. Try it now!

This section provides an overview of Morph’s protocol architecture, including:
- The Decentralized Sequencer Network
- Optimistic zkEVM & Responsive Validity Proof (RVP)
- Morph's Modular Architecture
- General Protocol Design

2-morph-modular-design.md:

---
title: Morph Modular Design
lang: en-US
keywords: [morph,ethereum,rollup,layer2,validity proof,optimistic zk-rollup]
description: Upgrade your blockchain experience with Morph - the secure decentralized, cost0efficient, and high-performing optimistic zk-rollup solution. Try it now!

The modular design of blockchain technology, known for its improved composability, has become a prevalent trend. Morph leverages this design principle to enhance its architecture and functionality.

!arichitecture

Overview

A modular design typically divides a Layer 1 blockchain into four core functions:

1. Consensus
2. Execution
3. Data Availability
4. Settlements

Morph applies this modular approach to its Layer 2 solution by dividing it into

three primary modules, each responsible for specific functionalities.

3 Major Morph Modules

Sequencer Network - Consensus & Execution

!Sequencer Network

Sequencer network responsible for the execution & consensus of the Layer 2 transactions, for more details please refer to Morph's decentralized sequencers.

Optimistic zkEVM - Settlement

!Optimistic zkEVM

State verification ensures that state changes on Layer 2 are valid on Layer 1. Morph introduces Optimistic zkEVM, a hybrid solution combining zk-rollups and optimistic rollups for state verification. The process involves a Morph innovation known as Responsive Validity Proof (RVP). This innovative approach finalizes and settles Layer 2 transactions and states efficiently. For more details, refer to the documentation on Responsive Validity Proof.

Rollup - Data Availability

!Rollup

The Rollup process involves submitting Layer 2 transactions and states to Layer 1, ensuring data availability. Morph's rollup strategy maximizes efficiency by compressing block content using zk-proofs, which helps manage the cost of Layer 1 data availability.

5 Morph Roles

Sequencers

Sequencers play a crucial role in the network by:

- Receiving Layer 2 user transactions and forming blocks.
- Reaching consensus with other sequencers.
- Executing blocks and applying state transitions.
- Batching blocks and submitting them to Layer 1.
- Synchronizing blocks with full nodes.
- Generating validity proofs when challenged.

Prover

Provers are essential for generating zk proofs when a sequencer is challenged. They synchronize Layer 2 transaction information and produce the necessary zk proofs to validate state changes.

Validator

Validators can be any user and play a key role in ensuring the correctness of states submitted by sequencers to Layer 1. They maintain an L2 node to synchronize transactions and state changes, triggering challenges when incorrect states are identified.

Nodes

Nodes facilitate easier access to Layer 2 transactions and states without actively participating in network operations. Running an L2 node is open to anyone and does not require permission.

Layer 1

Every Layer 2 solution relies on a Layer 1 blockchain for final settlements and data availability. For Morph, this role is fulfilled by Ethereum. Key contracts on Layer 1 ensure the security and finality of Layer 2 transactions and states.

6 Morph Components

L2 Node

The L2 node is central to Morph's architecture, interacting with various modules and roles. It includes sub-components such as:
- Transactions Manager (Mempool): Manages all Layer 2 transactions, accepting and storing user-initiated transactions.
- Executor: Applies state transitions and maintains the real-time status of Layer 2.
- Synchronizer: Synchronizes data between L2 nodes to restore network status.

Batch Submitter
The Batch Submitter is part of the sequencer, responsible for continuously obtaining L2 blocks, packaging them into batches, and assembling the batches into Layer 1 transactions, which are then submitted to the Layer 1 contract.

Consensus Client
Each sequencer runs a consensus client to reach consensus with other sequencers. The current design uses the Tendermint client to ensure seamless integration and developer friendliness.

zkEVM
zkEVM is part of the Prover and is a zk-friendly virtual machine used to generate zk proofs for Ethereum blocks and state changes. These zk proofs are ultimately used to prove the validity of L2 transactions and states.

Aggregators
Aggregators work with zkEVM to reduce the cost of verifying zk proofs by aggregating them for block production.

Layer 1 Contract
These contracts on Ethereum store Layer 2 transactions, execute global state changes, and bridge assets and information between Layer 2 and Layer 1. They also manage the election and governance of the sequencer set, inheriting the security of Ethereum.

Integration of Components, Roles, and Modules

!modular

The components form the foundation of the various roles in Morph. For instance, running an L2 node allows one to become a Node, while adding batch submitter and consensus client functionalities enables the role of Sequencer. These roles collaborate to perform the core functions of Morph, creating a complete and efficient rollup solution.

3-optimistic-zkevm.md:

---
title: Optimistic zkEVM
lang: en-US
keywords: [morph,layer2,validity proof,optimistic zk-rollup]
description: Upgrade your blockchain experience with Morph - the secure decentralized, cost0efficient, and high-performing optimistic zk-rollup solution. Try it now!

!RVP

Introduction to State Verification

Layer 2 state verification traditionally falls into two categories: fraud proofs and validity proofs. Morph introduces a new verification method called Responsive Validity Proof (RVP), combining the benefits of both approaches to address their limitations. Fraud proofs, while effective, suffer from capital inefficiency and low security assumptions. Additionally, no Optimistic Rollup (OP-Rollup) has fully implemented a permissionless fraud-proof challenge mechanism. Conversely, validity proofs offer high security but face practical issues with cost and efficiency that hinder Rollup scalability.

The Problem with Optimistic Rollups

In this model, Layer 2 (L2) optimistically assumes that the state changes submitted by the sequencer are valid without actively verifying their authenticity. Instead, a challenge period is introduced before the state changes are confirmed on Layer 1 (L1). During this period, external challengers verify the sequencer's submissions based on their own synchronized network status. If they find discrepancies, challengers can trigger a challenge process on L1 to prevent incorrect states from being confirmed.

Challenge Mechanism: Although all optimistic rollups claim to implement fraud proofs, only Arbitrum has successfully deployed them on the mainnet. Furthermore, the challengers are often limited to several whitelisted addresses. Fraud proofs in current optimistic rollup projects can be categorized into two types:

Non-Interactive Fraud Proofs: When a new state submitted by the sequencer is challenged, L1 re-executes all corresponding L2 transactions to generate a valid state for comparison with the state submitted by the sequencer. This process incurs significant gas costs and may lead to discrepancies between L2 and L1, as some transactions might produce different outcomes on L2 compared to L1, or L1 might not be able to execute certain L2 transactions. Optimism (OP) once used this approach but abandoned it due to these issues.

Interactive Fraud Proofs: To address the issues of non-interactive fraud proofs, multi-round interactive fraud proofs were introduced. This method involves determining the specific instruction execution that caused the incorrectness through multiple rounds of interaction between the sequencer and the challenger, then confirming fraud by executing the corresponding instructions on L1. This approach reduces computational costs and diminishes the issue of incongruent outcomes between L1 and L2. However, it introduces complexities, such as:
- Higher implementation difficulty
- Longer challenge periods (sufficient time must be reserved for complex interactions)
- Increased standards, impacting challengers' motivation

Currently, only several OP Rollups have implemented a complete interactive fraud proof mechanism on its mainnet among optimistic rollup projects. In contrast, several ZK-rollup projects have already launched on the mainnet.

These complexities highlight the need for improvement in existing optimistic

rollup models. Hence, Morph introduces the Responsive Validity Proof (RVP).

What is RVP?

!RVP

Responsive Validity Proof (RVP) integrates ZK-based validity proofs into the optimistic rollup framework. The process is as follows:

When challengers detect that the sequencer has submitted incorrect data, they initiate a challenge request to the sequencer on Layer 1 (L1). The sequencer must then generate the corresponding Zero-Knowledge (ZK) proof within a specified time (challenge period) and pass the verification of the L1 contract. If the verification passes, the challenge fails; otherwise, the challenge succeeds. This process combines the benefits of optimistic rollups and ZK-rollups, providing a balanced approach to security and efficiency.

Advantages of RVP Compared to Interactive Fraud Proofs

1. Shorter Challenge Period: RVP can reduce the challenge period from the typical 7 days to just 1-3 days, improving overall efficiency and user experience.
2. Reduced L2 Submission Costs: By using validity proofs, Layer 2 (L2) does not need to include most transaction bytes, significantly lowering submission costs.
3. Improved Challenger Experience: With RVP, challengers only need to initiate the challenge. The sequencer must prove their correctness by generating and verifying the corresponding ZK-proof, simplifying the challenger's responsibilities.
4. Seamless Transition to ZK-Rollup: The architectural design of RVP allows for an easy transition to a complete ZK-rollup. The primary change required is adjusting the sequencer's ZK-proof submission methods from responsive to active.

RVP enhances the optimistic rollup model by incorporating ZK-proofs, offering a more efficient, cost-effective, and secure solution. It addresses the limitations of traditional fraud proofs and paves the way for a seamless transition to full ZK-rollup implementations in the future.

How Can RVP Shorten the Challenge Period of an Optimistic Rollup?

The Need for a Challenge Period

Optimistic rollups incorporate a challenge period (or withdrawal period) to ensure that any malicious submissions by the sequencer can be identified and contested. This period provides sufficient time for challengers to verify transactions, conduct fraud proofs, and complete the challenge process, thereby ensuring that only valid state changes are confirmed on Layer 1 (L1).

Two main factors influence the length of the challenge period:

1. Completion Time: The time required for both parties to complete the challenge process.
2. Mitigating Malicious Behavior: Ensuring that there is enough time to address any attempts by sequencers to maliciously block the challenger’s transactions on L1.

Solutions to Shorten the Challenge Period

Concise and Direct Challenge Process: For multi-round interactive fraud proofs, the entire challenge process might require several rounds of interaction, each demanding significant time. For example, if the process requires 10 rounds, at least 20 blocks of time are needed to complete the challenge, considering the back-and-forth responses.

In contrast, RVP simplifies the challenge process by requiring only one interaction: the sequencer uploads the ZK-proof of the batch, which is then verified on L1. This streamlined process addresses the main problem of whether challengers have enough time to detect and prove incorrectness, thus

significantly reducing the challenge period.

Protection Against Malicious Behavior: In interactive fraud-proof systems, the challenged party might attempt to interfere with the challenge progress, such as launching a DoS attack on L1 to prevent challengers from interacting with L1 and submitting proofs.
With RVP, challengers only need to trigger the challenge. Once the challenge is initiated, the sequencer has no opportunity to interfere. The sequencer must then prove the correctness of its submission through the ZK-proof. This ensures that the normal challenge process is not affected by malicious behavior, further shortening the challenge period.

Key Benefits of RVP in Reducing the Challenge Period
- Efficiency: The single interaction required for RVP simplifies the challenge process, reducing the time needed for resolution.
- Security: By relying on ZK-proofs, RVP provides a robust mechanism to validate state changes without lengthy interactions.
- Cost-Effectiveness: The reduction in the number of interactions lowers the gas costs associated with challenge processes on L1.

By addressing these factors, RVP effectively shortens the challenge period from the traditional 7 days to just 1-3 days, offering a more efficient and secure solution for optimistic rollups.

Why is the Operating Cost Lower for L2 Based on RVP?

Compression of Transactions

In ZK-rollups, the validity of each transaction is confirmed through a submitted ZK-proof, which eliminates the need to include extensive transaction details. For example, the length of an Ethereum transaction is approximately 110 bytes, with the signature occupying around 68 bytes. In optimistic rollups, because transactions need to be replayed on L1, these signatures must be included to ensure validity. This increases the cost.
However, ZK-rollups only need to retain basic transaction information because the validity proof covers the entire batch. This compression capability reduces the amount of data that needs to be submitted to L1, significantly lowering costs.

Efficient Data Submission

RVP utilizes ZK-proofs to validate transactions, adopting the ZK-rollup advantage of transaction compression during batch data submission. This reduces the overall data volume and associated costs. Additionally, when there are no challenges, the sequencer does not incur the cost of generating and submitting ZK-proofs, further lowering operational expenses.

Comparison with Existing Solutions

The design of RVP ensures that the cost of rollup operations is lower than that of both existing optimistic rollups and traditional ZK-rollups. This efficiency is achieved by:
- Reducing the need for detailed transaction replays on L1.
- Leveraging ZK-proofs only when necessary, minimizing unnecessary proof generation costs.

RVP is Friendly to Challengers
The core of RVP is the use of validity proofs to ultimately validate challenged data. This benefits challengers in the following ways:
1. Simplified Challenge Process:
- In RVP, the sequencer is responsible for generating and verifying proofs. Challengers only need to initiate a challenge through staking, reducing the complexity and burden on challengers.
- This contrasts with traditional fraud proofs, where challengers must interact

multiple times with the sequencer, making the process cumbersome and complex.
2. Lower Threshold for Challengers:
- In many Layer 2 projects, sequencers have a high incentive to act maliciously due to potential high returns. Conversely, challengers typically see lower direct benefits, leading to a lack of motivation to challenge fraudulent transactions.
- RVP lowers the threshold for challengers by shifting the responsibility of proof generation to the sequencers, thus increasing the likelihood of detecting and correcting fraudulent behavior.
3. Mitigating Malicious Challenges:
- While there is a risk of challengers initiating unnecessary challenges to increase costs for sequencers, RVP mitigates this by requiring challengers to compensate sequencers for the costs incurred if a challenge is unsuccessful.
- This mechanism discourages frivolous challenges and ensures that only legitimate disputes are raised.

By adopting these strategies, RVP ensures a fairer and more efficient process for validating Layer 2 transactions, ultimately lowering operating costs and enhancing the security and integrity of the network.

Why do sequencers have to take on the responsibility of submitting ZK-proofs?

Some proposals have suggested that challengers could demonstrate the falsehood of a sequencer's submission by providing their own submission and corresponding ZK-proof. The two submissions could then be compared to identify any fraudulent activity by the sequencer. However, there are significant concerns with this approach:

Challengers would need to generate ZK-proofs using the transactions provided by the sequencer. If the sequencer submits invalid transactions, challengers cannot create ZK-proofs that can be authenticated on Layer 1 (L1). Therefore, it is more effective for sequencers to prove the correctness of their submissions. This approach ensures that the entity responsible for the transactions verifies their accuracy, maintaining the integrity of the system..
Why Not Simply Employ ZK-Rollups?

While verifying the validity of every state submission by the sequencer through numerous cryptographic calculations, as seen in current ZK-rollups, theoretically offers higher security, this approach presents several challenges:

The Cost of ZK-Rollup

Currently, projects such as zkSync and Polygon zkEVM have launched on the mainnet, showing that generating and verifying ZK-proofs is no longer the most pressing issue. However, these ZK-proofs still face cost and efficiency constraints. For instance, the average transaction cost on zkSync Era ranges from $0.51 to as high as $310, depending on L1 gas fees. This is significantly more expensive than the transaction costs of optimistic rollup projects like Arbitrum and Optimism. In contrast, with RVP, the high cost is avoided during normal network operation by only compressing transaction data using ZK-proofs when challenged. Normal operation incurs minimal costs, maintaining efficiency and affordability.

Block Finalization Time in ZK-Rollups

Theoretically, ZK-rollups should have no withdrawal period because the entire L2 state transition verification process through ZK-proof should be completed in minutes or even seconds. However, the practical reality is different. Due to technical limitations, the time required for final verification of ZK-proofs on L1 is much slower than expected. For example, zkSync Era takes about 20-24 hours for L2 blocks to be finalized, which is not significantly different from the optimized withdrawal periods of optimistic rollups.

Seamless Transition with RVP-Based Rollups

L2 scaling solutions incorporating RVP technology can be designed using the ZK-rollup framework, allowing for an easy transition from RVP-based L2 to standard ZK-rollup L2 as ZK technology matures. The primary adjustment needed is changing the sequencer's ZK-proof submission methods from responsive to active. Thus, RVP-based systems can seamlessly adopt full ZK-rollup benefits in the future.

1-morph-decentralized-sequencer-network.md:

---
title: Morph's Decentralized Sequencer Network
lang: en-US
keywords: [morph,ethereum,rollup,layer2,validity proof,optimistic zk-rollup]
description: Upgrade your blockchain experience with Morph - the secure decentralized, cost0efficient, and high-performing optimistic zk-rollup solution. Try it now!

!RVP

The Importance of Decentralized Sequencers

What is a sequencer and what does it do?

In a traditional Layer 1 blockchain, transactions are packaged and processed by miners in proof-of-work systems or validators nodes in proof-of-stake systems. These entities earn the authority to package, sequence, and produce blocks either through the competitive task of computational mining or via staking-based elections.

However, many current Layer 2 designs employ a single role, unburdened by competition or staking costs, responsible for packaging and sequencing all Layer 2 transactions. This entity is known as the “sequencer”. Its duties extend beyond sequencing; it is also tasked with generating L2 blocks, periodically committing Layer 2 transactions and state changes to Layer 1, and addressing any potential challenges to its submissions.

Centralized sequencers present a challenge due to their sole dominion over the sequencing and packaging of Layer 2 transactions. This monopoly raises concerns, largely stemming from this centralized control.

What are the problems with centralized sequencers?

Vulnerability of a Single Point of Failure

The proper functioning of Layer 2 is intrinsically tied to the operation of the sequencer. If the sequencer stops working, transactions from all Layer 2 users will not be processed, effectively bringing down Layer 2 operations. The problem is magnified when a single entity controls the sequencer. Should the entity fail, the entirety of the Layer 2 is paralyzed, rendering the system vulnerable to a single point of failure. Therefore, centralized sequencers pose a significant risk to the stability of Layer 2.

Excessive Transaction Censorship

Centralized sequencers have the ability to reject user-submitted transactions, rendering them unprocessable — a blatant form of transaction censorship. In a scenario where a centralized Layer 2 deliberately blocks transactions involving

its governance tokens, panic and selling among users is likely to follow.
Some solutions allow users to submit their intended transactions directly on Layer 1. However, this process is time-consuming, often taking several hours, and burdens users with Layer 1 gas fees. Therefore, this alternative does not fundamentally solve the problem.
In a decentralized sequencer framework, should one sequencer decline a transaction, users can still relay it to alternative sequencers. The content of the next block is ultimately determined through consensus, ensuring no single entity can censor transactions based on personal interests.

Monopoly Over MEV

Because the sequencer can determine the order (or "sequence") of received transactions, it effectively has a monopoly over all Miner Extractable Value (MEV). In this scenario, users must bear any potential losses incurred by the sequencer’s exclusive control over MEV, necessitating an additional and unwarranted layer of trust in the sequencer.
Decentralized sequencers introduce a competitive dynamic among multiple entities aiming for MEV. This competition eliminates the monopoly of any single sequencer, mitigating the adverse effects of unchecked MEV on users.

What's Morph's Approach to Decentralized Sequencers?

Morph is distinct from other Rollup projects due to the emphasis on establishing a decentralized sequencer network from inception. This design is guided by the following core principles:

Efficiency:
Morph is first and foremost an Ethereum scaling solution, focused on improved efficiency and cost reduction. Our solution must guarantee fast execution and transaction confirmation at Layer 2 while maintaining the highest possible level of decentralization.

Scalable and Manageable:
The sequencer network’s design prioritizes ease of maintenance, expansion, and updating. If one network functionality requires maintenance, it should not disrupt the operation of other functionalities. In addition, the sequencer network should be adaptable and easily upgradable as new and more efficient solutions emerge.

Solutions Formulated on These Principles
With these principles, Morph’s sequencer network design includes:
- Modularity: The structure emphasizes modular components that are loosely connected, allowing for swift upgrades or replacements.
- Byzantine Fault Tolerant (BFT) Consensus: Sequencers employ a BFT consensus for L2 block generation.
- BLS Signature for Batch Signing: Sequencers sign a collective of L2 blocks using the BLS signature method. The L1 contract then verifies this L2 consensus through the BLS signature.

:::tip
Why BLS signature?

A current basic signature algorithm such as ECDSA in Ethereum has an excessive cost. This issue arises because the signature data needs to be submitted to the Layer 1 contract and requires payment of the corresponding cost. As the number of validators increases, this cost will also increase proportionally. By using BLS signatures, the cost of uploading signatures can be maintained at a constant level, unaffected by the gradual growth of the sequencer's quantity.

:::

Architecture

The following is a simple illustration of Morph’s decentralized sequencing network architecture.

Sequencer Set Selection

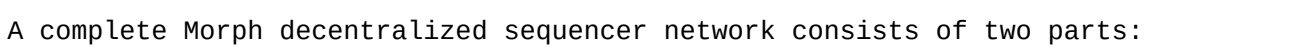
A complete Morph decentralized sequencer network consists of two parts:

- Sequencer Set : This forms the core group that provides sequencing services
- Sequencer Staking Contract: This contract facilitates the selection of the sequencer set via an election process.

Through the sequencer staking contract, members are elected into the sequencer set, where they collaboratively provide services for the Morph network. Periodically, the election results are synchronized to the Layer 1 Rollup contract. This synchronized data is utilized to obtain the BLS aggregate signatures of sequencer network participants for comparison and verification.

Layer 2 Blocks Generation

Given Morph's modular design, each sequencer operates a consensus client that runs BFT to communicate with other sequencers.



Following the BFT consensus protocol, the selected sequencer extracts transactions from the mempool, constructs blocks, and synchronizes these blocks with other sequencers to undergo verification and voting. The end result is the generation of new Layer 2 blocks.

Batching

Considering the costs of uploading to and validating signatures on Layer 1, sequencers will sign a batch of blocks with BLS signatures at designated checkpoints.

Post-signing, the designated sequencer forwards the collective batch of blocks to Layer 1 through its batch submitter component.

Consensus Verification

The selected sequencer must submit to the Layer 1 contract:

- The aggregated BLS signatures
- The batch of transactions
- The consensus-determined state

The Layer 1 contract will verify the submitted signature to confirm the transaction's consensus.

Summary

- Morph operates a native decentralized sequencer network based on BFT consensus.
- Through protocol and network optimization, Morph maximizes the scalability of Ethereum while ensuring decentralization.
- Based on BLS signatures, other participants in Layer 1 and Layer 2 can effectively verify the consensus results of Layer 2, ensuring the security provided by the sequencer network is confirmable at the Layer 1 level.

Roadmap

Stage 1: Close test on morph beta testnet

Stage 2: Decentralized sequencer network live on mainnet

Stage 3: Open election of sequencer set

Stage 4: Open Morph's sequencer network to the public l2 space

2-morph-staking-system-design.md:

---
title: Morph's Staking System Design
lang: en-US
keywords: [morph,ethereum,rollup,layer2,validity proof,optimistic zk-rollup]
description: Upgrade your blockchain experience with Morph - the secure decentralized, cost0efficient, and high-performing optimistic zk-rollup solution. Try it now!

What is Morph Staking System?

:::tip
Currently the Staking System is in beta testing phase, the design described in this document will change as the testing process progresses, and does not represent the final experience on the Mainnet.
:::

Morph Staking is a complete economic and engineering system built upon the decentralized sequencer network to ensure the operation and security of the network.

It can be divided into 2 parts:

ETH Staking

On Ethereum, potential sequencers are required to stake ETH in the layer 1 staking contract to become a staker first.

The ETH staking serves to increase the cost of malicious behavior by sequencers.

In case of confirmed dishonesty or negligence by a sequencer, the staked ETH will be slashed. The required ETH staking amount is immutable.

Morph token Staking

Morph token is the governance token of Morph (Gas token is ETH). In the staking system, it will play the following roles:

Staker is elected as the sequencer according to the amount of Morph tokens that token holders have delegated to them. So stakers need to attract Morph token holders to delegate their tokens to them. Only sequencers can receive network rewards.

Morph token holders can delegate stake their tokens to any stakers, which will determine whether stakers can be selected as sequencer. Sequencers receive rewards from the inflation of the Morph token based on the sequencer’s contribution, and delegators share a portion of the rewards based on their delegation amount.

Roles within the Staking System:

1. Staker (Sequencer Candidate): Anyone (required in the whitelist in the early stage) can stake ETH to L1 staking contract and become a staker. Only stakers can join the sequencer election.
2. Sequencer: Sequencers are able to perform the sequencer tasks and get reward from it. Sequencers are selected according to the delegation amount of Morph tokens.
3. Delegator: Morph token holders can delegate stake their tokens to any of the stakers. Delegators can share the rewards gained by delegated sequencer based on the delegation amounts.

Details of sequencer set election:

The determination of the sequencer set will be based on two points:

1. Sequencers must have staked a fixed amount of ETH in Layer 1 staking contract.
2. Assuming the maximum size of the sequencer set is X, based on the delegation amount of Morph token, select up to X sequencers from all valid candidates as sequencer set.

The sequencer set will be updated in real-time based on the above principles.

When there's a new MorphToken stake, the L2 staking contract will check if this would cause the sequencer set to change, and update the sequencer set if needed.

Joining the sequencer set means that all sequencers will have the right to participate in the current network operations to earn rewards, while also bearing the responsibility of maintaining the network's efficient and normal operation.

In practice, each sequencer, regardless of the delegation amount, has the same weight.

Sequencers can exit at any time. They need to submit an exit request on Layer 1 staking contract, then enter a lock-up period. After Layer 2 contracts complete the exit process and reach the unlocking block height, they are unlocked and could claim the staked ETH.

Rewards & Slash

Rewards

There are 3 potential rewards for sequencers within the Morph ecosystem.

L2 MorphToken staking rewards.

The Morph token is inflationary that increases 6% of the initial max total supply each year as the L2 Morph token staking rewards.

These 6% will be distributed everyday (one day is an epoch) to all the current running sequencers.

Sequencers will take commission first and distribute the rest to the delegators based on their delegation amount.

L2 Gas Income:

Sequencers take ETH from layer 2 users (Layer2 income) and spend ETH to submit batches to layer1 (Layer1 cost).

If the Layer1 cost is less than Layer2 income, the remaining value theoretically becomes the profit of Layer 2.

At the very beginning, the network collects all the Layer2 incomes and pays ETH to sequencers to cover their Layer 1 cost. In the future, we'll have a more detailed plan about how this part of the funds will be utilized.

ETH Re-staking yield:

To improve capital efficiency, we plan to leverage staker's ETH deposit to generate yield in restaking products, and the yield will still be allocated to stakers.

How to decide each sequencer's MorphToken rewards?

This is based on the block production records.

In each epoch, the total reward received by each sequencer and their delegators is calculated as follows:

sequencerreward = (sequencerproducedblock / totalproducedblocks) totalmorphtokeninflation

The sequencer rewards are eventually distributed to delegators (although sequencers can be their own delegators too). Sequencer can set a commission rate to take a share from it and the rate is adjustable.

sequencercommission = sequencerreward commissionrate

The reward each delegator of this sequencer receives is the remaining portion multiplied by the percentage of their staked amount:

delegatorreward = (sequencerreward - sequencercommission) delegationamount / totaldelegationamount

The user's delegation rewards will be calculated starting from the next epoch after the user stakes.

Example:

If today's (this epoch's) total Morph Inflation is 100, and there are 100 blocks produced in this epoch.

Sequencer A produced 10 blocks within this epoch, so he will receive:

sequencerreward = (sequencerproducedblock / totalproducedblocks) totalmorphtokeninflation = (10/100) 100 = 10 MorphToken.

If sequencer's commission rate is 5%

sequencercommission = sequencerreward commissionrate = 10 0.05 = 5 MorphToken

If one delegator staked 100 MorphToken and then there are a total 1000 MorphToken delegated to the sequencer.

delegatorreward = (sequencerreward - sequencercommission) delegationamount / totaldelegationamount = (100 - 5) (100/1000) = 9.5 MorphToken

Ideally, since the weights for each sequencer is the same, each sequencer will be able to produce the same amount of blocks within a certain epoch. Thus their rewards should be the same.

However, if the sequencer failed to perform their duties, the block production will be much lower thus their rewards would be much lower too.

Slash

Based on the optimistic zkEVM design, there will be validators constantly verifying the batch submitted by sequencers, and if they think sequencers committed fraud, validators will start a challenge through the L1 rollup contract.

Read more about the challenge here:

To prevent fraudulent behavior by Sequencers from affecting network security, the following rules need are established:

- Validators will challenge a fixed batch, and all sequencers who signed that batch will collectively be challenged.
- When a sequencer takes on the role of batch submitter, repeated instances of timeouts accumulating to a certain extent will result in a deduction of rewards or removal from the sequencer set (not fully implemented yet).
- Sequencers with long periods that have not produced blocks will be removed from the sequencer set (not fully implemented yet).

:::tip
The submitter rotation and submission timeout is part of decentralized rollup design, you can read the details here.
:::

For the reward and slash functionalities, we have 2 contracts:

- L2 Record Contract: The off-chain data affecting rewards and penalties will be collected and recorded in the L2 Record contract through an Oracle, primarily consisting of rollup data and Block data.
- L2 Distribute Contract: Sequencers and Delegator will manually claim rewards based on the Record.

Governance:

We have a governance contract right now that decides some of the network parameters. Currently, only sequencers can create proposals and vote.

In the next phase of the roadmap, we are planning to build a complete governance system that allows all Morph token holders to decide every aspect of the network.

Major Process

Staking & Sequencer Selection

Morph token staking will be divided into 2 stages based on the network status:

- Phase 1: Morph token inflation and staking rewards not started yet. Sequencer elections will be FCFS at the beginning, but delegate stake is allowed. No morph token rewards since there is no new token generation yet.

- Phase 2: Morph token inflation and staking rewards is started.
The sequencer election will officially start and based on the delegation amount of Morph token, the rewards will start to be distributed too.

How is the sequencer set generated?

1. L1 Staking Contract: Add potential sequencers to the whitelist.
2. L1 Staking Contract: Potential sequencer will be able to register and stake eth on Ethereum to become eligible for sequencer election (become staker).
3. An add staker message will be sent as a cross-layer message from L1 staking contract to L2 staking contract.
4. L2 Staking Contract: Will update stakers with the message synced.
5. L2 Staking Contract: Users will be able to delegate/undelegate stake MorphToken to a staker.
6. L2 Sequencer Contract: L2 staking contract will update the sequencer set by calling L2 sequencer contact based on the ranking of the Morph token delegation amount, the top staker will be elected as sequencer.

Sequencer network consensus & Verification on Layer 1

- Every submitted batch requires the BLS signature of more than 2/3 of the sequencers within the sequencer set to be accepted by the L1 rollup contract.

Notice: Currently, the BLS 12-381 signature pre-compiled contract has not been implemented on Ethereum. Therefore, the L1 rollup contract cannot verify whether the batch is signed by the L2 sequencer set.
Until this functionality is available, the rollup contracts only allow batch submission from stakers included in the whitelist. This measure is in place to enable us to slash the ETH deposit in case of fraudulent submissions. After signature verification is implemented, the submitter will be permissionless and the sequencers which signed the fraudulent batch will be slashed instead of the submitter.

1. L1 Rollup Contract: Batch submitter commits the batch to rollup contract.
2. L1 Rollup Contract: Rollup contract verifies the batch’s BLS signature and compares it with the sequencer set sync from L1 staking contract. It will only accept the batch if the verification passed.

Slash for Sequencers

What happens if validators successfully challenge sequencers?

- Sequencer will be slashed all staked ETH and removed from sequencer set if challenger succeeds.
- Even if get proven fraud by multiple challenges, each sequencer will only be slashed once.
- The challenger reward for a successful challenge is a fixed proportion of the staking amount.
- If the slash makes all the sequencers go down, then the L2 will stop running. We can restart by upgrading the L1 staking contract, reset stakers and sequencer sets. This does not affect the Layer 2 state as no transactions will be processed because of this.

Process:

1. L1 Staking Contract: Slash staked ETH of sequencers who signed the fraud batch and remove them from sequencer set.
2. L1 Staking Contract: Distribute validator rewards.
3. A remove staker message will be sent as a cross-layer message from L1 staking contract to L2 staking contract.
4. L2Staking Contract: Update sequencer set.

Delegation Stake

1. L2 Staking Contract: Staker set delegation commission rate by their own will.
2. L2 Oracle: Upload sequencers work records (block production records, submitter work records, expect work records) on the epoch basis (an epoch is a day).
3. L2 MorphToken Contract: Mint MorphToken (inflation) as delegation reward and sent to L2 distributor contract.
4. L2 Staking Contract: Users claim delegation reward, sequencers claim commission.

Staker/Sequencer exit

The exit lock period should be long enough to ensure that stakers and sequencers in L2 have been updated and greater than the challenge period of sequencer’s last produced block (if staker is also sequencer).

1. L1 Staking Contract: Stakers apply to exit, the stake ETH is locked to enter the lock period.
2. A remove staker message will be sent as a cross-layer message from L1 staking contract to L2 staking contract.
3. L2 Staking Contract: Received the message, remove staker, and sequencers (if the staker is also sequencer).
4. L1 Staking Contract: Withdraw allowed until reach unlock block height, remove staker info after claiming.

1-rollup.md:

---
title: Rollup
lang: en-US
keywords: [morph,ethereum,rollup]
description: Explain how rollup process works in Morph

:::info

As the foundation of a Layer 2 project, the "Rollup" process refers to the method by which Layer 2 assembles L2 transactions and state into batches and subsequently submits them to L1, along with the L2 state.

Within Morph's architecture, this Rollup process is executed by the Batch Submitter components.

:::

An overview of Morph Rollup Design:

!rollup

Constructing the Batch

The L2 Node within the sequencer generates L2 blocks based on consensus results and updates the local state of L2. The batch submitter must query the L2 node to retrieve the latest L2 blocks.

The batch submitter then reconstructs L2 blocks, compiling:

- Transactions: All transactions contained within the blocks.
- Blockinfo : Essential information from each block.

The batch submitter continues fetching and reconstructing blocks until it processes a block with a BLS signature, indicating the batch point has been reached. The reconstructed block data is used to construct a batch, which contains:

- lastBlocknumber : The number of the final block in the batch.
- Transactions : Encoded transactions within the batch.
- BlockWitness : Encoded blockinfos, utilized for zkProof.
- PreStateRoot : The stateRoot before the batch is applied.
- PostStateRoot : The stateRoot after the batch is applied.
- WithdrawalRoot : L2 withdrawal Merkle tree root.
- Signature : The batch’s BLS signature.

:::info

Blockinfo (BlockWitness) is needed since Morph employs zk technology to prove the accuracy of submitted batch data. It serves as a witness in the Zero-Knowledge Proof.

:::

Putting Multiple Batches into a Single Rollup Transaction

While it's standard for roll-up projects to include only one batch per L1 roll-up transaction, Morph optimizes by inserting as many batches as feasible into a single L1 transaction.
This efficiency-driven approach significantly reduces overall costs, as the L1 fee is a predominant component of the transaction costs associated with the L2. By optimizing the utilization of available space, Morph achieves cost-effectiveness without compromising transaction integrity.

Submitting Batch Data to the Rollup Contract

The batch submitter will eventually send an Ethereum transaction from its L1 account to Morph's main contract.

The transaction's calldata contains the batch data.

:::info

Based on the development process of ERC-4337, future batch data will likely be incorporated into a new ‘blob’ structure to further decrease costs.

:::

Once the transaction is submitted and confirmed on Ethereum, validator nodes can reconstruct and verify the validity of sequencers' submissions using the transactional data within the batch.

Finalize the batches

If batches are valid according to Morph's responsive validity proof standards, all transactions within the batches will be finalized, including withdrawal transactions.

Consequently, withdrawal requests will be fulfilled, and the corresponding locked assets on Layer 1 will be released.

Decentralize Batch Submitter

What is Batch Submitter?

A Batch Submitter plays a crucial role in the "rollup" process, acting as the bridge that connects Layer 2 (L2) data with Ethereum (Layer 1 or L1). Their primary responsibilities include:
- Collecting L2 transactions and block data, assembling them into a cohesive batch.
- Embedding this batch data within a Layer 1 transaction.
- Executing this transaction by calling the Layer 1 contract to complete the rollup process.

!rollup

What is the relationship between Sequencers & Batch Submitters?

The Batch submitter function is often integrated within the broader 'sequencer' role. In a decentralized sequencer network architecture, each sequencer is equipped with or has access to a batch submitter component. This integration is key to achieving and maintaining the highest levels of decentralization.

This structure ensures that the data uploaded to Layer 1 remains decentralized, preventing a single entity from controlling the rollup process.

How to decentralize the Batch Submitter?

To uphold the aforementioned principles, it is essential to ensure that multiple sequencers can share rollup tasks evenly within the same time frame. Our approach to achieving this involves a rotation system for sequencers to take turns with the responsibility of calling the batch submitter, as detailed below:

Submitter Rotation

- Epoch Cycle Role Switching: Sequencers alternate roles as batch submitters within an established Epoch cycle.
- Cross-Epoch Execution Capability: Any Sequencer can perform a Rollup for another Sequencer's Epoch.
- Timeout Logging: The system records instances when not a single rollup happens during an epoch, the epoch will be marked as “timeout” as well as the responsible sequencer.

Timeout

- Timeout Identification: If an epoch passes without a rollup (batch submission), it's identified as a "timeout." The timing of a rollup is pegged to the confirmation time of the Layer 1 rollup transaction.

- Epoch Rotation: The duration of an epoch and the rotation schedule are governed by the sequencer network governance. Sequencers are assigned indexes which determine their responsibility for an epoch. With changes in the sequencer set, indexes are reassigned, and epochs rotate accordingly based on these new assignments.

Penalties for Timeout
- Accumulated Penalties: Sequencers that frequently exhibit timeout behaviors may face penalties that related to their Layer 1 ETH staking, if the timeout

records reaches to a certain level, sequencer may/will be slashed from the sequencer network.

Module Design

Below you can find the contracts that are responsible for each module and their responsibilities:

Layer1
- RollupContract: records the rollup executor and sync with L2

Layer2
- SequencerContract: Sync Sequencers
- GovContract: Manage Batch & Epoch Parameters
- SubmitterContract:
  - Record Epoch information
  - Record Rollup history
  - Record Submitter Workload
  - Record Submitter Timeouts
- IncentiveContract: Conducts Incentive and Penalty actions

2-communicate-between-morph-and-ethereum.md:

---
title: Communication between Morph and Ethereum
lang: en-US
keywords: [morph,ethereum,rollup,layer2,validity proof,optimistic zk-rollup]
description: Upgrade your blockchain experience with Morph - the secure decentralized, cost0efficient, and high-performing optimistic zk-rollup solution. Try it now!

Although Morph is a Layer 2 solution built atop Ethereum, it remains a separate and distinct blockchain. Thus, it’s essential to establish a communication channel between Morph and Ethereum to facilitate the smooth transfer of assets and messages. The communication can occur in two directions: from Ethereum to Morph and from Morph to Ethereum.

The Basics of Morph - Ethereum Bridge

Transferring assets between Ethereum and Morph involves the following process:

- Asset Locking and Wrapping: To initiate the transfer, a user must lock their asset on the cross-layer bridge. When the lock is confirmed, Morph mints a Wrapped Token that represents the value of the locked asset, in a procedure referred to as a "deposit".

- Asset Reception: Following the minting, the user or intended recipient will receive the asset on Morph, reflecting the value of the originally locked asset.

- Reverse Process: Conversely, to transfer assets back to Ethereum, the bridge can unlock the original asset on Ethereum by burning the Wrapped Token on Morph. This is referred to as "withdraw".

Furthermore, the bridge’s utility extends beyond asset transfers. It employs the same foundational principle for message transfers, enabling the conveyance of data payloads across two network layers.

Understanding the Gateway

The Gateway serves as the primary entry point for users to interact with the entire bridge system. While the core process of transferring assets across layers still relies on message transmission, we recommend using the Gateway approach for efficient cross-layer transactions.

Catering to diverse cross-layer asset transfer needs, we have designed distinct Gateways such as the ETH Gateway, standard ERC20 Gateway, and others.

Furthermore, we have implemented the Gateway Router to call on different Gateways based on the type of assets you have. This facilitates seamless interaction with the Gateway Router contract.

L1 Gateway Contract	Description
L1GatewayRouter	The gateway router supports the deposit of ETH and ERC20 tokens.
L1ETHGateway	The gateway to deposit ETH.
L1StandardERC20Gateway	The gateway for standard ERC20 token deposits.
L1CustomERC20Gateway	The gateway for custom ERC20 token deposits.
L1WETHGateway	The gateway for Wrapped ETH deposits.

L2 Gateway Contract	Description
L2GatewayRouter	The gateway router supports the withdraw of ETH and ERC20 tokens.
L2ETHGateway	The gateway to withdraw ETH.
L2StandardERC20Gateway	The gateway for standard ERC20 token withdraw.
L2CustomERC20Gateway	The gateway for custom ERC20 token withdraw.
L2WETHGateway	The gateway for Wrapped ETH withdraw.

Deposit (L1 to L2 message)

!Deposit Process

Constructing a Deposit Request Through the Gateway

A bridge request, whether it is for ETH, ERC20, or ERC721, is essentially a cross-layer message, which necessitates the initial construction of a message.

Generally, the message structure remains consistent, especially for ETH & ERC20 Gateways.

Employing a token gateway compiles a standard token gateway message and relays it to the CrossDomainMessenger.

Passing the Message Through the CrossDomainMessenger

The CrossDomainMessenger serves as the core unit of cross-layer communication, with corresponding messenger contracts on both Layer 1 and Layer 2.

For a deposit, the L1 messenger sends a message to the L2 messenger, akin to a contract call on Layer 1, which means custom messages (contract interactions) can be constructed to perform various types of cross-layer interactions.

Executing the Message on Layer 2

The cross-domain message is delivered to the L1MessageQueueWithGasPriceOracle, which then triggers an event called QueueTransaction.

The Sequencer will monitor this event and include a Layer 2 transaction in its next block.

How to make sure Sequencer doesn't fake a deposit transaction?

Sequencers may have the motivation to forge a non-existent deposit transaction, such as minting a large amount of Layer 2 tokens and transferring these to an address they own.

Morph prevents these risks with two measures:

Due to Morph’s decentralized Sequencer architecture, forging transactions would require control of at least two-thirds of the Sequencers, a challenging feat.

Morph's optimistic zkEVM framework allows challengers to detect such malicious behavior and initiate challenges to correct the misconduct.

A Layer 2 executor, holding the cross-layer message, interacts with the L2 messenger to execute the message, which may include transferring L2 ETH or ERC20 tokens to the recipient.

Finalizing the Deposit Message

The completion of the deposit process involves more than just executing the request on Layer 2. There is a possibility that the Layer 2 execution and its corresponding state update could be reverted due to incorrect batch data being identified through the challenge process.

Therefore, a deposit request is only considered complete once the corresponding batch of the deposit execution transaction is finalized.

Typically, this follows a simple workflow:

- The deposit execution transactions are compiled into a batch and submitted to Layer 1 by batch submitters.

- Following the challenge period, valid batches are finalized by subsequent batch submissions using rollup.commitBatch.

- During finalization, the L1MessageQueueAndGasPriceOracle removes(pop) the deposit message from the queue, marking the completion of the deposit process.

Withdraw (L2 -> L1 message)

!Withdraw Process

Finalizing a Withdrawal

Unlike Deposits, a withdrawal request must undergo 2 processes for execution:

1. Prove that a withdraw request actually happened on Layer 2 by verifying a Merkle tree proof against the withdrawal tree root committed by sequencers.

2. Wait for the challenge period to end and finalize the withdraw tree root, addressing the risk of sequencer submitting incorrect batch data, including the withdraw tree root.

Typically, these 2 processes happen at the same time. Once the withdraw tree root is finalized, users can call the proveAndRelayMessage method within the L1CrossDomainMessenger contract to execute the withdrawal message.

```
solidity
function proveAndRelayMessage(
        address from,
        address to,
        uint256 value,
        uint256 nonce,
        bytes memory message,
        bytes32[32] calldata withdrawalProof,
        bytes32 withdrawalRoot
    )
```

This function serves two primary purposes:

1. It checks if the withdraw tree root associated with this message has been finalized through the rollup contract.
2. It verifies whether the withdraw request actually occurred by validating the provided Merkle proof.

Upon successful completion of both processes, this method will execute the corresponding action, such as releasing the user's ETH on Layer 1 for a standard ETH withdrawal request.

Understanding the Withdraw Tree

Withdrawal actions involve interacting with L1 assets/contracts as a result of a Layer 2 transaction. Consequently, it’s imperative to verify the existence of a Layer 2 transaction that triggers a withdrawal request, in a manner that is verifiable on Layer 1.

To achieve this, we introduce a structure known as a Withdraw Tree, which records every L2 withdrawal transaction within a Merkle tree. Thus, a Merkle tree's properties can be leveraged to confirm a withdrawal request’s occurrence.

The term Withdraw Tree refers to an append-only Sparse Merkle Tree (SMT) with leaf nodes that capture information on assets being transferred out of the network.
Each leaf in the Withdraw Tree, known as a Withdraw leaf, falls into two categories: type 0 for recording asset(s) information and type 1 for recording messaging information.

A withdraw leaf, in particular, is a Keccak256 hash of the ABI encoded packed structure with cross domain message.

The Withdraw Tree is instrumental in cataloging withdrawal transactions and

ascertaining the legitimacy of withdrawal requests.

Morph has pre-deployed a Simple Merkle Tree contract dedicated to constructing the Layer 2 withdraw tree.

This tree incorporates three methods:

1. getTreeroot - Retrieves the current tree's root hash.
2. appendMessageHash - Appends a new leaf node to the tree.
3. verifyMerkleProof - Verifies if a leaf node exists in the tree, indicating the validity of the bridge request it represents.

Understanding the Challenge Period & Batch Finalization

The Optimistic zkEVM architecture mandates that each L2 transaction be submitted to Layer 1 and undergo a challenge period before finalization.

This process is vital to validate the Layer 2 state, eventually validating the authenticity of the withdraw request.

The withdraw tree root, integral for withdrawal request verification, is also submitted by sequencers once the challenge period, batches, and states have been finalized.

Cross-layer (Bridge) Errors

With the design of cross-layer bridges, the cross-layer message for deposits needs to be executed and have its Layer 2 states updated. Sending a cross-chain request successfully does not guarantee its successful execution on L2.

Prior to this, there is a possibility of the cross-layer message failing during execution on Layer 2.
This section outlines the potential scenarios and solutions for handling failed cross-layer deposit messages.

Cross-layer (Bridge) Failure Scenarios:
Two primary types of failures can occur in cross-layer (bridge) communications:

1. Gas Failure: Cross-layer messages sent from the L1 to the L2 might fail during execution on the L2 due to limitations in gasLimit or code logic.

2. Skipped Message: Some data executions may trigger overflows in the circuits of L2 nodes, leading to the omission or skipping of cross-layer messages.

Handling Cross-layer (Bridge) Failures:

For Gas Failures:

- When the L1CrossDomainMessenger contract on L1 dispatches a cross-layer message, it records the corresponding message hash but does not incorporate the gasLimit in this record. Post-execution on L2, the L2CrossDomainMessenger performs an equivalent calculation, storing the contract call result in mapping(isL1MessageExecuted). This measure prevents multiple executions of the same message and facilitates the adjustment of gasLimit parameters for replaying failed messages.

- Replay Message: If gasLimit is insufficient, causing a failed execution on the L2, a new cross-layer message with a revised gasLimit parameter can be sent by calling L1CrossDomainMessenger.replayMessage.

For Skipped Messages:

- Messages dropped due to potential circuit overflow on the L2 are skipped and not executed. Custom cross-layer calling contracts need to implement the

onDropMessage method to consider such cases.

- The gateway contract includes the onDropMessage method, designed to refund the initiator of the cross-layer message.

- Calling L1CrossDomainMessenger.dropMessage discards the cross-layer message and triggers the onDropMessage method in the originating contract, passing the transaction's value and message as msg.value and method parameters, accordingly.

4-transactions-life-cycle.md:

---
title: Transactions Life Cycle
lang: en-US
keywords: [morph,ethereum,rollup,layer2,validity proof,optimistic zk-rollup]
description: Upgrade your blockchain experience with Morph - the secure decentralized, cost0efficient, and high-performing optimistic zk-rollup solution. Try it now!

How is a L2 transaction processed on Morph

1. Submit Transaction

User-initiated transactions are first sent to the L2 node's mempool, where they await selection and processing by a sequencer.

2. Transaction Consensus

Within the sequencer network, transactions undergo a consensus process. A selected sequencer proposes a block containing the transaction and then send the blocks to the consensus layer (consensus client).

Other sequencers then validate this block by executing this block, effectively verifying the transaction's legitimacy.

3. Transaction Execution

Once all the votes on the block are finalized, each sequencer & node will apply this block to update its local state.

4. Transaction Batching

When batch point is reached, each sequencer will need to construct the batch with all the blocks consensussed for the last epoch, the batch will go through consensus by requiring each sequencer to sign, all of the signature will be aggregated with BLS signature.

5. Batch Sequencing

The selected sequencer will commit the batches to the Layer 1 Rollup contract for both verification and to ensure data availability.

6. Batch Verification

A batch (so do the transactions within the batch) will first go through the BLS signature verification by the rollup contract to confirm the L2 consensus results, and then a batch will go through a challenge period to be marked as finalized, solidifying their status within the L1 and L2 state.

Morph Transaction Status

Processing

Once submitted, a transaction enters the consensus phase managed by sequencers and is placed into a block pre-execution.

Confirmed

Post-execution by the Sequencer, the transaction’s updated state is local to L2. It is then batched and sent to L1, where it must undergo a challenge period before finalization.

Safe

The batch that contains the transaction is submitted to Layer 1 but not finalized yet.

Finalized

A transaction is considered finalized after it survives the challenge period or is verified by a Zero-Knowledge Proof (ZK-Proof). Only then is it officially integrated into the final L1 and L2 state.

5-difference-between-ethereum-and-morph.md:

---
title: Difference between Morph and Ethereum
lang: en-US
keywords: [morph,ethereum,rollup,layer2,validity proof,optimistic zk-rollup]
description: Upgrade your blockchain experience with Morph - the secure decentralized, cost0efficient, and high-performing optimistic zk-rollup solution. Try it now!

There are several technical differences between Ethereum’s EVM and Morph's optimistic zkEVM. We’ve compiled a list to help you understand these distinctions better.

:::tip
For most Solidity developers, these technical details won't significantly impact your development experience.
:::

EVM Opcodes

Opcode	Solidity equivalent	Morph Behavior
BLOCKHASH	block.blockhash	Returns keccak(chainid || blocknumber) for the last 256 blocks.
COINBASE	block.coinbase	Returns the pre-deployed fee vault contract address. See Contracts.
DIFFICULTY / PREVRANDAO	block.difficulty	Returns 0.
BASEFEE	block.basefee	Disabled. If the opcode is encountered, the transaction will be reverted.
SELFDESTRUCT	selfdestruct	Disabled. If the opcode is encountered, the transaction will be reverted.

EVM Precompiles

The RIPEMD-160 (address 0x3) blake2f (address 0x9), and point evaluation (address 0x0a) precompiles are currently not supported. Calls to unsupported precompiled contracts will revert. We plan to enable these precompiles in future hard forks.

The modexp precompile is supported but only supports inputs of size less than or equal to 32 bytes (i.e. u256).

The ecPairing precompile is supported, but the number of points(sets, pairs) is limited to 4, instead of 6.

The other EVM precompiles are all supported: ecRecover, identity, ecAdd, ecMul.

Precompile Limits

Because of a bounded size of the zkEVM circuits, there is an upper limit on the number of calls that can be made for some precompiles. These transactions will not revert, but simply be skipped by the sequencer if they cannot fit into the space of the circuit.

Precompile / Opcode	Limit
keccak256	3157
ecRecover	119
modexp	23
ecAdd	50
ecMul	50
ecPairing	2

:::tip Dencun upgrade opcode not available

Opcodes from the Dencun upgrade are not yet available on Morph, including MCOPY, TSTORE, TLOAD, BLOBHASH and BLOBBASEFEE. Additionally, EIP-4788 for accessing the Beacon Chain block root is not supported. We recommend using shanghai as your EVM target and avoiding using a Solidity version higher than 0.8.23.

:::

State Account

Additional Fields

We added two fields in the current StateAccount object: PoseidonCodehash and CodeSize.

```
go
type StateAccount struct {
      Nonce    uint64
      Balance  big.Int
      Root     common.Hash // merkle root of the storage trie
      KeccakCodeHash []byte // still the Keccak codehash
      // added fields
      PoseidonCodeHash []byte // the Poseidon codehash
      CodeSize uint64
}
```

CodeHash

Related to this, we maintain two types of codehash for each contract bytecode: Keccak hash and Poseidon hash.

KeccakCodeHash is kept to maintain compatibility for EXTCODEHASH.

PoseidonCodeHash is used for verifying the correctness of bytecodes loaded in the zkEVM, where Poseidon hashing is far more efficient.

CodeSize

When verifying EXTCODESIZE, it is expensive to load the whole contract data into the zkEVM. Instead, we store the contract size in storage during contract creation. This way, we do not need to load the code — a storage proof is sufficient to verify this opcode.

Block Time

:::tip Block Time Subject to Change

Blocks are produced every second, with an empty block generated if there are no transactions for 5 seconds. However, this frequency may change in the future.
:::

To compare, Ethereum has a block time of 12 seconds.

Reasons for Faster Block Time in Morph
User Experience:

- A faster, consistent block time provides quicker feedback, enhancing the user experience.

- Optimization: As we refine the zkEVM circuits in our testnets, we can achieve higher throughput than Ethereum, even with a smaller gas limit per block or batch.

Notice:
- TIMESTAMP will return the timestamp of the block. It will update every second.
- BLOCKNUMBER will return an actual block number. It will update every second. The one-to-one mapping between blocks and transactions will no longer apply.

<!--
We also introduce the concept of system transactions that are created by the op-node, and are used to execute deposits and update the L2's view of L1. They have the following attributes:

- Every block will contain at least one system transaction called the L1 attributes deposited transaction. It will always be the first transaction in the block.
- Some blocks will contain one or more user-deposited transactions.
- All system transactions have an EIP-2718-compatible transaction type of 0x7E.
- All system transactions are unsigned, and set their v, r, and s fields to null.

:::Warning Known Issue
Some Ethereum client libraries, such as Web3js, cannot parse the null signature fields described above. To work around this issue, you will need to manually filter out the system transactions before passing them to the library.
:::
-->

Future EIPs

Morph closely monitors emerging Ethereum Improvement Proposals (EIPs) and adopts them when suitable. For more specifics, join our community forum or Discord for

discussions.

<!-- EVM Target version

To avoid unexpected behaviors in your contracts, we recommend using ‘london’ as the target version when compiling your smart contracts.

You can read in more details on Shanghai hard fork differences from London on the Ethereum Execution spec and how the new PUSH0 instruction impacts the Solidity compiler.
-->

1-welcome-to-morph.md:

---
title: Welcome to Morph!
lang: en-US
keywords: [morph,ethereum,rollup,layer2,validity proof,optimistic zk-rollup]
description: Upgrade your blockchain experience with Morph - the secure decentralized, cost0efficient, and high-performing optimistic zk-rollup solution. Try it now!

Thank you for choosing Morph! We're excited to have you onboard with the first optimistic ZK-EVM scaling solution for Ethereum. Morph is designed to meet a variety of needs, and you can easily navigate to the most relevant information for your interests.

:::tip

Beta Testnet Stage: Morph is currently in the beta testnet phase, offering a brand-new platform for exploration. We encourage you to delve into its features and capabilities.

:::

Web3 Enthusiasts: If you are a web3 enthusiast who wants to try out the Morph, you can start with how to connect to Morph.

Researchers: For those seeking in-depth understanding of Morph's unique offerings compared to other solutions, the how Morph works section is your go-to resource for comprehensive details.

Developers: As a skilled developer ready to build on Morph, the developer documentation provides all necessary resources and guides to kickstart your development journey.

Looking for help

Having issues while developing or exploring? Join our discord channel and talk to us in the right channel. We would love to hear your thoughts or feedback on how we can improve your experience, too.

2-wallet-setup.md:

---

title: Wallet Setup
lang: en-US
keywords: [morph,ethereum,rollup,layer2,validity proof,optimistic zk-rollup]
description: Upgrade your blockchain experience with Morph - the secure decentralized, cost0efficient, and high-performing optimistic zk-rollup solution. Try it now!

Wallet

To interact with dApps on Morph, you need a compatible wallet. Below are some example wallets and configuration tips.

<!--
Bitget Wallet

TBD
-->

MetaMask

- Installation: MetaMask can be installed from their official website.
- Importing Configurations: To set up MetaMask for Morph Testnet, click the "add to wallet" button on the https://explorer-holesky.morphl2.io/. This will automatically import the chain ID and RPC URLs for the Morph Testnet.
- Using Ethereum Holesky Testnet: Morph Testnet utilizes the Ethereum Holesky testnet as its underlying L1, which is already configured in MetaMask by default. To access it, enable "Show/hide test networks" in the MetaMask network selection dropdown.

Manual network configuration

Currently, the Add to wallet links may not be compatible with all wallets yet. If you are having issues using them, you may need to manually add the Holesky Testnet and Morph by inserting the configuration details from the table below:

Network Configuration

Name	RPC Url(s)	Chain ID	Block explorer	Symbol
Morph Holesky Testnet	https://rpc-quicknode-holesky.morphl2.io	2810	https://explorer-holesky.morphl2.io	ETH
Ethereum Holesky	https://ethereum-holesky-rpc.publicnode.com/	17000	https://holesky.etherscan.io	ETH

You can also visit chainlist to add Morph testnet and Ethereum testnet.

3-faucet.md:

---
title: Faucet
lang: en-US
keywords: [morph,ethereum,rollup,layer2,validity proof,optimistic zk-rollup]
description: Upgrade your blockchain experience with Morph - the secure decentralized, cost0efficient, and high-performing optimistic zk-rollup solution. Try it now!

Faucet

Morph Holesky Website Faucet

!website faucet

Our website faucet is live!

Users can claim Morph ETH & USDT to fund their initial activities.

3rd Party Ethereum Holesky ETH Faucet

To use Morph's public testnet, obtain testnet ETH on Holesky, then bridge it to the Morph testnet.

Here are a few Holesky faucet apps:

https://stakely.io/en/faucet/ethereum-holesky-testnet-eth

https://faucet.quicknode.com/ethereum/holesky

https://holesky-faucet.pk910.de/

https://cloud.google.com/application/web3/faucet/ethereum (needs a Google account)

Once you receive ETH on Holesky, you should see it in your wallet on the Holesky Network.

It may take a few seconds for them to appear, but you can check the status by looking for a transaction to your address on the Holesky Block Explorer.

Discord Morph Holesky Faucet

Morph Holesky ETH

You can obtain the Morph Holesky ETH in our discord too for development purposes.

Using the /morpheth command and type your address will grant you 0.01 Morph Holesky ETH.

Once succeed, you will see the following message:

!success

ERC20 USDT

:::tip

Currently, we set the limit that for each discord user, you can only request the tokens once every 24 hours.

:::

You can obtain morph's version of USDT on Holesky through our discord faucet, here's how it works:

1. Join our discord server through this link.

2. Find the #| discord-faucet channel.

3. Type /faucet in the channel and add your Holesky address behind it.

!command

4. Wait for a few seconds.

5. Once succeeded, you will see this in the channel.

!success

6. Add the Morph Holesky USDT information to your wallet.

Ethereum Holesky USDT address: "0xD6e9Cd5ef382b0830653d1b2007D5Ca6987FaA26"

Morph Holesky USDT address: "0x9E12AD42c4E4d2acFBADE01a96446e48e6764B98"

7. Check you wallet for USDT balance and start to bridge!

4-bridge.md:

---
title: Bridge
lang: en-US
keywords: [morph,ethereum,rollup,layer2,validity proof,optimistic zk-rollup]
description: Upgrade your blockchain experience with Morph - the secure decentralized, cost0efficient, and high-performing optimistic zk-rollup solution. Try it now!

Deposit from Holesky to Morph Testnet

Instructions:

:::tip Use the bridge here

 https://bridge-holesky.morphl2.io

:::

1. Open your MetaMask wallet and switch to the Holesky network.

!image1
!image1

2. Within Morph’s Bridge app, click Connect wallet, select MetaMask, and approve the connection if prompted.

!image2

3. Make sure that Holesky is selected under ‘From’ and Morph L2 under ‘To’. If not, click the "↓" button to switch their positions.

4. Select the token that you want to transfer.

5. Click the Send button to initiate the deposit.

:::tip
If this is your first time transferring an ERC20 token, you need to approve the Holesky Bridge contract to access your ERC20 token.
:::

6. A window will pop up asking for confirmation of the transfer transaction, click Deposit.

!image3

7. Click the Confirm button in MetaMask. Once the transfer transaction is finalized, the token will be deducted from your Holesky wallet address.

!image5

8. While you wait, you can check status of your transactions by clicking on the transactions button.

!image6

How long does it take for a token to arrive to Morph Testnet ?

A token transfer from Holesky to Morph Testnet may take 8 to 14 minutes (time for block to become Safe on Holesky) before it appears in your Morph wallet. To check the progress of your deposit transactions, follow these steps:

1. Click your wallet address at the top-right corner of the Bridge web app.

!image6

2. Click on Transactions. A pop-up panel will display your recent transactions.

:::tip
Note: For deposit transactions (L1 -> L2), once your transaction is confirmed as Safe on Holesky (8 to 14 minutes), you will see a Success status. Your funds will then be relayed to L2.
:::

!image8

3. Click on the most recent Holesky transaction hash.

!image9

4. You will be taken to a Transaction Details page in the Explorer. Verify your transaction status (this transaction is confirmed on Holesky).

!image10

5. Once your transaction status shows success on L2, return to the Bridge app to see a transaction hash and funds in your Morph L2 wallet.

!image11

!image12

Withdraw from Morph Testnet to Holesky

To withdraw funds from Morph Testnet, follow these steps:
1. Initiate the withdrawal on Morph Testnet.
2. Wait for the withdrawal root to be published on L1 (Holesky). This usually takes a few minutes, but it may take longer during outages.
3. Prove withdrawal.
4. Wait for the verification challenge period, which lasts seven days from the time the withdrawal is proven on L1 (Holesky).
5. Claim your withdrawal.

Initiate withdrawal

1. Click Connect Wallet and select MetaMask. If prompted, approve the connection in your wallet.

2. Select Withdraw. Choose the asset and amount you wish to withdraw.

!image13

3. Click Send ETH to Holesky.

!image14

4. Click Initiate withdrawal, wait for a few minutes to confirm. After it is finished, you need to switch the network in your wallet and then prove the withdrawal on Holesky.

!image15

!image16

5. Waiting for the batch submission to be completed.

!image17

!image18

Waiting for the verification challenge period

1. Click your address in the top right corner.

2. Click Transactions and then Withdrawals. This will display a list of your recent withdrawals and their status. Or you can find a notice in the top area, by clicking the button View Account (see the pic below).

!image19

!image20

!image21

3. You can search for the transaction hash on Morph Explorer.

!image22

!image23

4. Click the L1 State Root Submission Tx to see when the transaction was written to L1 (Holesky).

!image24

!image25

Claim the Withdrawal

1. Once the challenge period is over, the status will change to Claim.

2. Click Claim withdrawal.

!image26

3. Confirm the withdrawal in the wallet.

!image27

4. Wait until the withdrawal is completed.

!image28

1-morph-points.md:

---
title: Morph Points
lang: en-US
keywords: [morph,ethereum,rollup,layer2,validity proof,optimistic zk-rollup]
description: Upgrade your blockchain experience with Morph - the secure decentralized, cost0efficient, and high-performing optimistic zk-rollup solution. Try it now!

To bring our community even closer to the action, we're excited to announce the Morph Zoo.

 - a multiphase program designed to engage, reward, and celebrate the vibrant ecosystem within our testnet and, eventually, our mainnet. Our aim is to immerse you in our world, one where consumer needs are prioritized and where your input directly impacts our growth trajectory.

Morph Points are used to measure your engagement and contributions within the Morph ecosystem with a variety of campaigns which will be listed in Morph Zoo. They represent your active participation and achievements in our world.

Morph Zoo's Season 1: The Genesis Jungle was officially launched on May 13, 2024.

How Do I Earn Morph Points?

To earn Morph points, you can participate in the following activities within Season 1:
1. Engage with our ecosystem DApps. The more you do, the more you earn!
2. Check in daily to earn voting power, which can then be used to vote for your favorite projects. Voting earns you Morph Points!
3. Lastly, you can earn additional points by participating in events organized by Morph partners.

Where will Morph Points be displayed?

Morph Points for season 1 will be recorded on the "My Points" page of the official campaign site. However, please note that Morph Points earned through dapp engagement will be calculated and displayed at the conclusion of the event, along with the final tally of all points earned through all activities.

What can I do with Morph Points?

Morph points will play an important role when taking into account future airdrops and other exclusive rewards. Stay tuned for more!

Can I transfer Morph Points?

No. Morph Points are linked to specific users and wallets with no way to transfer.
It's time to explore Morph's Genesis Jungle, discovering special dapps while earning generous Morph Points as you do!

2-voting-rules.md:

---
title: Voting Rules in Voice of the Jungle
lang: en-US
keywords: [morph,ethereum,rollup,layer2,validity proof,optimistic zk-rollup]
description: Upgrade your blockchain experience with Morph - the secure decentralized, cost0efficient, and high-performing optimistic zk-rollup solution. Try it now!

Make your voice heard in our second Jungle activity: Voice of the Jungle. Use Voting power to vote for your favorite DApps, your votes help shape the ecosystem, make your 'roar' heard and you will be rewarded with Morph Points.

What is voting power?

To enhance our ecosystem and community, we've introduced a voting program to attract more valuable projects and DApps to the Morph ecosystem. Earn Voting Power through daily check-ins and unlocking bonus Mystery Boxes. Remember, it is the act of spending your Voting Power that earns you Morph Points, not the amount of unused Voting Power. At the end of the event, your Moph Points rewards will be calculated based on how much Voting Power you utilized, so be sure to vote every day!

How can I use my Voting Power?

You can utilize your Voting Power on the voting page to support projects that you like. You can vote for different projects and earn Points for the projects you vote for.

Voting rules:

1）Acquire Voting Power in two ways: daily check-ins and bonus Mystery Boxes.

2) To record your Voting Power and voting activities, you must sign in with your Web3 wallet. This verifies your Voting Power, without granting access to your assets.

3) You can allocate up to 100 Voting Power to each individual project per day. If you wish to vote more, you must wait until the next day.

4) Voting Power spent on a project is immediately consumed; it cannot be reused for multiple projects.

5) Only the act of voting will earn you Morph Points; simply accumulating Voting Power without using it will not earn you Morph Points for this activity.

Daily Check-in Rules:

Earn Voting Power by checking in daily. On the first day that you check-in, you will receive 2 Voting Power. On day two, you will receive 2 more Voting Power plus the same amount of Voting Power you received the previous day. After checking in for 7 days in a row, your Voting Power reward for that 7th day will be doubled to 28. So if you consistently check in for a week, you will receive 70 Voting Power total:

2(Day 1) + 4(Day 2) + 6(Day 3) + 8(Day 4) + 10(Day 5) + 12(Day 6) + 28(Day 7 you get a 2x multiplier bonus [142]) = 70

Please remember that on the 8th consecutive check-in day, your Voting Power reward will be reset back to 2, as this marks the beginning of a new 7-day cycle. However, to reward your commitment, on the 14th consecutive day, your bonus multiplier will be increased to 3x for that day. This means that on the fourteenth day, you actually receive 56 Voting Power. In fact, every consecutive day that you check in that lands on a multiple of 7, your bonus multiplier will increase by an additional factor compared to the previous week.

So if you consistently check in every day for a month or four weeks in a row, you will receive a total of 448 Voting Power:

(2+4+6+8+10+12)x4=168
The same increasing Voting Power for the first 6 days of each week
+
(28+56+84+112) = 280 The 7th day reward with an increasing multiplier each week
= 448

:::tip
Please note that forgetting to check in one day will disrupt the 7-day cycle resetting it completely. This means you will start over from day one if that happens. For instance, if you have consistently checked in for 2 days but forgot on the 3rd day, you will only receive 2 Voting Power on the 4th day instead of 8.
:::