How to use Timeboost

Timeboost is a new transaction ordering policy for Arbitrum chains. With Timeboost, anyone can bid for the right to access an express lane on the sequencer for faster transaction inclusion.

In this how-to, you'll learn how to bid for the right to use the express lane and submit transactions through the express lane. To learn more about Timeboost and the key terms used on this page, refer to thegentle introduction .

This how-to assumes that you're familiar with the following:

- How Timeboost works
- viem
- , since the snippets of code present in the how-to use this library

Note about transferring express lane rights Please note that in the initial release of Timeboost, transferring of express lane control via the either thesetTransferor or thetransferExpressLaneController will not be supported by the Arbitrum Nitro node software in the initial launch and may be implemented at a future date via a regular node upgrade. Calls made to these two functions on the auction contract will be successful, but actual transfer of the rights will not be recognized by the node software (including the sequencer).

A round's express lane controller, at their choice, can still send transactions signed by others on a per-transaction basis, as explained later in this guide.

How to submit bids for the right to be the express lane controller

To use the express lane for faster transaction inclusion, you must win an auction for the right to be the express lane controller for a specific round.

info Remember that, by default, each round lasts 60 seconds, and the auction for a specific round closes 15 seconds before the round starts. These default values can be configured on a chain using theroundDurationSeconds andauctionClosingSeconds parameters. Auctions are facilitated by an auction contract, and bids get submitted to an autonomous auctioneer that interact with the contract. Let's look at the process of submitting bids and finding out the winner of an auction.

Step 0: gather required information

Before we begin, make sure you have:

- Address of the auction contract
- Endpoint of the autonomous auctioneer

Step 1: deposit funds into the auction contract

Before bidding on an auction, we need to deposit funds in the auction contract. These funds are deposited in the form of the ERC-20 token used to bid, also known as thebidding token . We will be able to bid for an amount that is equal to or less than the tokens we have deposited in the auction contract.

To see the amount of tokens we have deposited in the auction contract, we can call the functionbalanceOf in the auction contract:

const depositedBalance =

await publicClient . readContract ({ address : auctionContractAddress , abi : auctionContractAbi , functionName :

'balanceOf' , args :

[userAddress] , }) ; console . log (Current balance of { userAddress } in auction contract: { depositedBalance }) ; If we want to deposit more funds to the auction contract, we first need to know what the bidding token is. To obtain the address of the bidding token, we can call the functionbiddingToken in the auction contract:

const biddingTokenContractAddress =

await publicClient . readContract ({ address : auctionContractAddress , abi : auctionContractAbi , functionName :

'biddingToken' , }) ; console . log (biddingToken: { biddingTokenContractAddress }) ; Bidding token in Arbitrum chains On Arbitrum One and Arbitrum Nova, the default bidding token is WETH. Once we know what the bidding token is, we can deposit funds to the auction contract by calling the functiondeposit of the contract after having it approved as spender of the amount we want to deposit:

// Approving spending tokens const approveHash =

await walletClient . writeContract ({ account , address : biddingTokenContractAddress , abi :

parseAbi (['function approve(address,uint256)']) , functionName :

'approve' , args :

[auctionContract , amountToDeposit] , }) ; console . log (Approve transaction sent: { approveHash }) ;

// Making the deposit const depositHash =

await walletClient . writeContract ({ account , address : auctionContractAddress , abi : auctionContractAbi , functionName :

'deposit' , args :

[amountToDeposit] , }) ; console . log (Deposit transaction sent: { depositHash }) ;

Step 2: submit bids

Once we have deposited funds into the auction contract, we can submit bids for the current auction round.

We can obtain the current round by calling the functioncurrentRound in the auction contract:

const currentRound =

await publicClient . readContract ({ address : auctionContractAddress , abi : auctionContractAbi , functionName :

'currentRound' , }) ; console . log (Current round: { currentRound }) ; The above shows the current round that's running. At the same time, the auction for the next round might be open. For example, if thecurrentRound is 10, the auction for round 11 is happening right now. To check whether or not that auction is open, we can call the functionisAuctionRoundClosed of the auction contract:

let currentAuctionRoundIsClosed =

await publicClient . readContract ({ address : auctionContractAddress , abi : auctionContractAbi , functionName :

'isAuctionRoundClosed' , }) ; Remember that, by default, auctions for a given round open 60 seconds before that round starts and close 15 seconds before the round starts, so there might be no auctions opened at certain times. Once we know what is the current round we can bid for (currentRound + 1) and we have verified that the auction is still open (!currentAuctionRoundIsClosed), we can submit a bid.

Bids are submitted to the autonomous auctioneer endpoint. We need to send aauctioneer_submitBid request with the following information:

- chain id
- address of the express lane controller candidate (for example, our address if we want to be the express lane controller)
- address of the auction contract
- round we are bidding for (in our example,currentRound + 1
-)
- amount in wei of the deposit ERC-20 token to bid
- signature (explained below)

Minimum reserve price The amount to bid must be above the minimum reserve price at the moment you are bidding. This parameter is configurable per chain. You can obtain the minimum reserve price by calling the methodminReservePrice()(uint256) in the auction contract. Let's see an example of a call to this RPC method:

const currentAuctionRound = currentRound +

1 ; const hexChainId :

0x { string }

=

0x { Number (publicClient . chain . id) . toString (16) } ;

const res =

await

fetch (< AUTONOMOUS_AUCTIONEER_ENDPOINT

, { method :

'POST' , headers :

{

'content-type' :

'application/json'

} , body :

JSON . stringify ({ jsonrpc :

'2.0' , id :

'submit-bid' , method :

'auctioneer_submitBid' , params :

[{ chainId : hexChainId , expressLaneController : userAddress , auctionContractAddress : auctionContractAddress , round :

0x { currentAuctionRound . toString (16) } , amount :

0x { Number (amountToBid) . toString (16) } , signature : signature , } ,] , }) , }); The signature that needs to be sent is an EIP-712 signature over the following typed structure data:

- Domain:Bid(uint64 round,address expressLaneController,uint256 amount)
- round
- : auction round number
- expressLaneController
- : address of the express lane controller candidate
- amount
- : amount to bid

Here's an example to produce that signature with viem:

const currentAuctionRound = currentRound +

1 ;

const signatureData =

hashTypedData ({ domain :

{ name :

'ExpressLaneAuction' , version :

'1' , chainId :

Number (publicClient . chain . id) , verifyingContract : auctionContractAddress , } , types :

{ Bid :

[{ name :

'round' , type :

'uint64'

} , { name :

'expressLaneController' , type :

'address'

} , { name :

'amount' , type :

'uint256'

} ,] , } , primaryType :

'Bid' , message :

{ round : currentAuctionRound , expressLaneController : userAddress , amount : amountToBid , } , }) ; const signature =

await account . sign ({ hash : signatureData , }) ; info You can also call the functiongetBidHash in the auction contract to obtain thesignatureData , specifying theround ,userAddress andamountToBid . When sending the request, the autonomous auctioneer will return an empty result with an HTTP status200 if received correctly. If the result returned contains an error message, it means that something went wrong. Following are some of the error messages that can help us understand what's happening:

Error Description MALFORMED_DATA wrong input data, failed to deserialize, missing certain fields, etc. NOT_DEPOSITOR the address is not an active depositor in the auction contract WRONG_CHAIN_ID wrong chain id for the target chain WRONG_SIGNATURE signature failed to verify BAD_ROUND_NUMBER incorrect round, such as one from the past RESERVE_PRICE_NOT_MET bid amount does not meet the minimum required reserve price on-chain INSUFFICIENT_BALANCE the bid amount specified in the request is higher than the deposit balance of the depositor in the contract

Step 3: find out the winner of the auction

After the auction closes and before the round starts, the autonomous auctioneer will call the auction contract with the two highest bids received so the contract can declare the winner and subtract the second-highest bid from the winner's deposited funds. After this, the contract will emit an event with the new express lane controller address.

We can use this event to determine whether or not we've won the auction. The event signature is:

event

SetExpressLaneController (uint64 round , address

indexed previousExpressLaneController , address

indexed newExpressLaneController , address

indexed transferor , uint64 startTimestamp , uint64 endTimestamp) ; Here's an example to get the log from the auction contract to determine the new express lane controller:

const fromBlock =

< any recent block ,

for example during the auction

const logs =

await publicClient . getLogs ({ address : auctionContractAddress , event : auctionContractAbi . filter ((abiEntry)

=> abiEntry . name

===

'SetExpressLaneController') [0] , fromBlock , }) ;

const newExpressLaneController = logs [0] . args . newExpressLaneController ; console . log (New express lane controller: { newExpressLaneController }) ; If you won the auction, congratulations! You are the express lane controller for the next round, which, by default, will start 15 seconds after the auction closes. The following section explains how we can submit a transaction to the express lane.

How to submit transactions to the express lane

The sequencer immediately sequences transactions sent to the express lane, while regular transactions are delayed 200ms by default. However, only the express lane controller can send transactions to the express lane. The previous section explained how to participate in the auction as the express lane controller for a given round.

The express lane is handled by the sequencer, so transactions are sent to the sequencer endpoint. We need to send atimeboost_sendExpressLaneTransaction request with the following information:

- chain id
- current round (following the example above,currentRound
-)
- address of the auction contract
- sequence number: a per-round nonce of express lane submissions, which is reset to 0 at the beginning of each round
- RLP encoded transaction payload
- conditional options for Arbitrum transactions (more information
-)
- signature (explained below)

Timeboost-ing third party transactions Notice that while the express lane controller must sign thetimeboost_sendExpressLaneTransaction request, the actual transaction to be executed can be signed by any party. In other words, the express lane controller can receive transactions signed by other parties and sign them to apply the time advantage offered by the express lane to those transactions. Support foreth_sendRawTransactionConditional Timeboost doesn't currently support theeth_sendRawTransactionConditional method. Let's see an example of a call to this RPC method:

const hexChainId :

0x { string }

=

0x { Number (publicClient . chain . id) . toString (16) } ;

const transaction =

await walletClient . prepareTransactionRequest (...) ; const serializedTransaction =

await walletClient . signTransaction (transaction) ;

const res =

await

fetch (< SEQUENCER_ENDPOINT

, { method :

'POST' , headers :

{

'content-type' :

'application/json'

} , body :

JSON . stringify ({ jsonrpc :

'2.0' , id :

'express-lane-tx' , method :

'timeboost_sendExpressLaneTransaction' , params :

[{ chainId : hexChainId , round :

0x { currentRound . toString (16) } , auctionContractAddress : auctionContractAddress , sequence :

0x { sequenceNumber . toString (16) } , transaction : serializedTransaction , options :

{ } , signature : signature , } ,] , }) , }); The signature that needs to be sent is an Ethereum signature over the bytes encoding of the following information:

- Hash ofkeccak256("TIMEBOOST_BID")
- Chain id in hexadecimal, padded to 32 bytes
- Auction contract address
- Round number in hexadecimal, padded to 8 bytes
- Sequence number in hexadecimal, padded to 8 bytes
- Serialized transaction

Here's an example to produce that signature:

const hexChainId :

0x { string }

=

0x { Number (publicClient . chain . id) . toString (16) } ;

const transaction =

await walletClient . prepareTransactionRequest (...) ; const serializedTransaction =

await walletClient . signTransaction (transaction) ;

const signatureData =

concat ([keccak256 (toHex ('TIMEBOOST_BID')) , pad (hexChainId) , auctionContract , toHex (numberToBytes (currentRound ,

{ size :

8

})) , toHex (numberToBytes (sequenceNumber ,

{ size :

8

})) , serializedTransaction ,]) ; const signature =

await account . signMessage ({ message :

{ raw : signatureData } , }) ; When sending the request, the sequencer will return an empty result with an HTTP status200 if it received it correctly. If the result returned contains an error message, something went wrong. Following are some of the error messages that can help us understand what's happening:

Error Description MALFORMED_DATA wrong input data, failed to deserialize, missing certain fields, etc. WRONG_CHAIN_ID wrong chain id for the target chain WRONG_SIGNATURE signature failed to verify BAD_ROUND_NUMBER incorrect round, such as one from the past NOT_EXPRESS_LANE_CONTROLLER the sender is not the express lane controller NO_ONCHAIN_CONTROLLER there is no defined, on-chain express lane controller for the round What happens if you're not the express lane controller? If you are not the express lane controller and you try to submit a transaction to the express lane, the sequencer will respond with the errorNOT_EXPRESS_LANE_CONTROLLER orNO_ONCHAIN_CONTROLLER .

How to withdraw funds deposited in the auction contract

Funds are deposited in the auction contract to have the right to bid in auctions. Withdrawing funds is possible through two steps: initiate withdrawal, wait for two rounds, and finalize withdrawal.

To initiate a withdrawal, we can call the functioninitiateWithdrawal in the auction contract:

const initWithdrawalTransaction =

await walletClient . writeContract ({ account , address : auctionContractAddress , abi : auctionContractAbi , functionName :

'initiateWithdrawal' , }) ; console . log (Initiate withdrawal transaction sent: { initWithdrawalTransaction }) ; This transaction will initiate a withdrawal of all funds deposited by the sender account. When executing it, the contract will emit aWithdrawalInitiated event, with the following structure:

event

WithdrawalInitiated (address

indexed account , uint256 withdrawalAmount , uint256 roundWithdrawable) ; In this event,account refers to the address whose funds are being withdrawn,withdrawalAmount refers to the amount being withdrawn from the contract, androundWithdrawable refers to the round at which the withdrawal can be finalized.

After two rounds have passed, we can call the methodfinalizeWithdrawal in the auction contract to finalize the withdrawal:

const finalizeWithdrawalTransaction =

await walletClient . writeContract ({ account , address : auctionContractAddress , abi : auctionContractAbi , functionName :

'finalizeWithdrawal' , }) ; console . log (Finalize withdrawal transaction sent: { finalizeWithdrawalTransaction }) ;

How to identify timeboosted transactions

Transactions sent to the express lane by the express lane controller and that have been executed (regardless of them being successful or having reverted) can be identified by looking at their receipts or the message broadcasted by the sequencer feed.

Transaction receipts include now a new fieldtimeboosted , which will betrue for timeboosted transactions, andfalse for regular non-timeboosted transactions. For example:

blockHash 0x56325449149b362d4ace3267681c3c90823f1e5c26ccc4df4386be023f563eb6 blockNumber 105169374 contractAddress cumulativeGasUsed 58213 effectiveGasPrice 100000000 from 0x193cA786e7C7CC67B6227391d739E41C43AF285f gasUsed 58213 logs [ ] logsBloom
0x000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
root status 1

(success) transactionHash 0x62ea458ad2bb408fab57d1a31aa282fe3324b2711e0d73f4777db6e34bc1bef5 transactionIndex 1 type

2 blobGasPrice blobGasUsed to 0x0000000000000000000000000000000000000001 gasUsedForL1 "0x85a5" l1BlockNumber "0x6e8b49" timeboosted true In the sequencer feed, theBroadcastFeedMessage struct now contains ablockMetadata field that represents whether a particular transaction in the block was timeboosted or not. The field blockMetadata is an array of bytes and it starts with a byte representing the version (0), followed byceil(N/8) number of bytes whereN is the number of transactions in the block. If a particular transaction was timeboosted, the bit representing its position in the block will be set to1 , while the rest will be set to0 . For example, if theblockMetadata of a particular message, viewed as bits is00000000 01100000 , then the 2nd and 3rd transactions in that block were timeboosted. Edit this page Last updatedonJan 31, 2025 Previous L1 Ethereum RPC providers Next Overview