How to bridge tokens via a custom gateway

PUBLIC PREVIEW DOCUMENT This document is currently inpublic preview and may change significantly as feedback is captured from readers like you. Click theRequest an update button at the top of this document orjoin the Arbitrum Discord to share your feedback. Do you really need a custom gateway? Before starting to implement and deploy a custom gateway, it is strongly encouraged to analyze the current solutions that Arbitrum's token bridge provides: thestandard gateway and thegeneric-custom gateway . These solutions provide enough functionality to solve the majority of bridging needs from projects. And if you are in doubt about your current approach, you can always ask for assistance on ourDiscord server . In this how-to you'll learn how to bridge your own token between Ethereum (Layer 1 or L1) and Arbitrum (Layer 2 or L2), using a custom gateway. For alternative ways of bridging tokens, don't forget to check out thisoverview .

Familiarity withArbitrum's token bridge system , smart contracts, and blockchain development is expected. If you're new to blockchain development, consider reviewing ourQuickstart: Build a dApp with Arbitrum (Solidity, Hardhat)before proceeding. We will useArbitrum's SDK throughout this how-to, although no prior knowledge is required.

We will go through all steps involved in the process. However, if you want to jump straight to the code, we have createdthis script in our tutorials repository that encapsulates the entire process.

Step 0: Review the prerequisites (a.k.a. do I really need a custom gateway?)

Before starting to implement and deploy a custom gateway, it is strongly encouraged to analyze the current solutions that Arbitrum's token bridge provides: thestandard gateway and thegeneric-custom gateway . These solutions provide enough functionality to solve the majority of bridging needs from projects. And if you are in doubt about your current approach, you can always ask for assistance on ourDiscord server .

Having said that, there are multiple prerequisites to keep in mind when deploying your own custom gateway.

First of all, theL1 counterpart of the gateway , must conform to theIL1ArbitrumGateway and theITokenGateway interfaces. This means that it must have, at least:

- A methodoutboundTransferCustomRefund
- , to handle forwarded calls fromL1GatewayRouter.outboundTransferCustomRefund
- . It should only allow calls from the router.
- A methodoutboundTransfer
- , to handle forwarded calls fromL1GatewayRouter.outboundTransfer
- . It should only allow calls from the router.
- A methodfinalizeInboundTransfer
- , to handle messages coming ONLY from L2's gateway.
- Two methodscalculateL2TokenAddress
- andgetOutboundCalldata
- to handle other bridging operations.
- Methods to send cross-chain messages through theinbox contract
- . An example implementation can be found insendTxToL2
- andsendTxToL2CustomRefund
- onL1ArbitrumMessenger
- .

Furthermore, if you plan on having permissionless registration of tokens in your gateway, your L1 gateway should also have aregisterCustomL2Token method, similar to the one being used in Arbitrum'sgeneric-custom gateway .

On the other hand, theL2 counterpart of the gateway , must conform to theITokenGateway interface. This means that it must have, at least:

- A methodoutboundTransfer
- , to handle external calls, and forwarded calls fromL2GatewayRouter.outboundTransfer
- .
- A methodfinalizeInboundTransfer
- , to handle messages coming ONLY from L1's gateway.
- Two methodscalculateL2TokenAddress
- andgetOutboundCalldata
- to handle other bridging operations.
- Methods to send cross-chain messages through theArbSys precompile
- . An example implementation can be found insendTxToL1
- onL2ArbitrumMessenger
- .

What about my custom tokens?

If you are deploying custom gateways, you will probably want to support your custom tokens on L1 and L2 too. They also have several requirements they must comply with. You can find more information about it inHow to bridge tokens via Arbitrum's generic-custom gateway .

Step 1: Create a gateway and deploy it on L1

This code is for testing purposes The code contained within the following sections is meant for testing purposes only and does not guarantee any level of security. It has not undergone any formal audit or security analysis, so it is not ready for production use. Please exercise caution and due diligence while using this code in any environment. We'll begin the process by creating our custom gateway and

deploying it on L1. A good example of a custom gateway isArbitrum's generic-custom gateway . It includes all methods required plus some more to support the wide variety of tokens that can be bridged through it.

In this case, we'll use a simpler approach. We'll create a gateway that supports only one token and has the ability to be disabled/enabled by the owner of the contract. It will also implement all necessary methods. To simplify the deployment process even further, we won't worry about setting the addresses of the counterpart gateway and the custom tokens at deployment time. Instead, we will use a functionsetTokenBridgeInformation that will be called by the owner of the contract to initialize the gateway.

// SPDX-License-Identifier: Apache-2.0 pragma

solidity

^ 0.8.0 ;

import

"./interfaces/ICustomGateway.sol" ; import

"./CrosschainMessenger.sol" ; import

"@openzeppelin/contracts/token/ERC20/IERC20.sol" ; import

"@openzeppelin/contracts/access/Ownable.sol" ;

/* * @title Example implementation of a custom gateway to be deployed on L1 * @dev Inheritance of Ownable is optional. In this case we use it to call the function setTokenBridgeInformation * and simplify the test / contract

L1CustomGateway

is IL1CustomGateway , L1CrosschainMessenger , Ownable {

// Token bridge state variables address

public l1CustomToken ; address

public l2CustomToken ; address

public l2Gateway ; address

public router ;

// Custom functionality bool

public allowsDeposits ;

/* * Contract constructor, sets the L1 router to be used in the contract's functions and calls L1CrosschainMessenger's constructor * @param router_ L1GatewayRouter address * @param inbox_ Inbox address / constructor (address router_ , address inbox_)

L1CrosschainMessenger (inbox_)

{ router = router_ ; allowsDeposits =

false ; }

/* * Sets the information needed to use the gateway. To simplify the process of testing, this function can be called once * by the owner of the contract to set these addresses. * @param l1CustomToken_ address of the custom token on L1 * @param l2CustomToken_ address of the custom token on L2 * @param l2Gateway_ address of the counterpart gateway (on L2) / function

setTokenBridgeInformation (address l1CustomToken_ , address l2CustomToken_ , address l2Gateway_)

public onlyOwner { require (l1CustomToken ==

address (0) ,

"Token bridge information already set") ; l1CustomToken = l1CustomToken_ ; l2CustomToken = l2CustomToken_ ; l2Gateway = l2Gateway_ ;

// Allows deposits after the information has been set allowsDeposits =

true ; }

/// @dev See {ICustomGateway-outboundTransfer} function

outboundTransfer (address l1Token , address to , uint256 amount , uint256 maxGas , uint256 gasPriceBid , bytes

calldata data)

public

payable override returns

(bytes

memory)

{ return

outboundTransferCustomRefund (l1Token , to , to , amount , maxGas , gasPriceBid , data) ; }

/// @dev See {IL1CustomGateway-outboundTransferCustomRefund} function

outboundTransferCustomRefund (address l1Token , address refundTo , address to , uint256 amount , uint256 maxGas , uint256 gasPriceBid , bytes

calldata data)

public

payable override returns

(bytes

memory res)

{ // Only execute if deposits are allowed require (allowsDeposits ==

true ,

"Deposits are currently disabled") ;

// Only allow calls from the router require (msg . sender == router ,

"Call not received from router") ;

// Only allow the custom token to be bridged through this gateway require (l1Token == l1CustomToken ,

"Token is not allowed through this gateway") ;

address

from ; uint256 seqNum ; { bytes

memory extraData ; uint256 maxSubmissionCost ; (from , maxSubmissionCost , extraData)

=

_parseOutboundData (data) ;

// The inboundEscrowAndCall functionality has been disabled, so no data is allowed require (extraData . length ==

0 ,

"EXTRA_DATA_DISABLED") ;

// Escrowing the tokens in the gateway IERC20 (l1Token) . transferFrom (from ,

address (this) , amount) ;

// We override the res field to save on the stack res =

getOutboundCalldata (l1Token ,

from , to , amount , extraData) ;

// Trigger the crosschain message seqNum =

_sendTxToL2CustomRefund (l2Gateway , refundTo , from , msg . value , 0 , maxSubmissionCost , maxGas , gasPriceBid , res) ; }

emit

DepositInitiated (l1Token ,

from , to , seqNum , amount) ; res = abi . encode (seqNum) ; }

/// @dev See {ICustomGateway-finalizeInboundTransfer} function

finalizeInboundTransfer (address l1Token , address

from , address to , uint256 amount , bytes

calldata data)

public

payable override onlyCounterpartGateway (l2Gateway)

{ // Only allow the custom token to be bridged through this gateway require (l1Token == l1CustomToken ,

"Token is not allowed through this gateway") ;

// Decoding exitNum (uint256 exitNum ,

)

= abi . decode (data ,

(uint256 ,

bytes)) ;

// Releasing the tokens in the gateway IERC20 (l1Token) . transfer (to , amount) ;

emit

WithdrawalFinalized (l1Token ,

from , to , exitNum , amount) ; }

/// @dev See {ICustomGateway-getOutboundCalldata} function

getOutboundCalldata (address l1Token , address

from , address to , uint256 amount , bytes

memory data)

public

pure override returns

(bytes

memory outboundCalldata)

{ bytes

memory emptyBytes =

"" ;

outboundCalldata

abi . encodeWithSelector (ICustomGateway . finalizeInboundTransfer . selector , l1Token , from , to , amount , abi . encode (emptyBytes , data)) ;

return outboundCalldata ; }

/// @dev See {ICustomGateway-calculateL2TokenAddress} function

calculateL2TokenAddress (address l1Token)

public

view override returns

(address)

{ if

(l1Token == l1CustomToken)

{ return l2CustomToken ; }

return

address (0) ; }

/// @dev See {ICustomGateway-counterpartGateway} function

counterpartGateway ( )

public

view override returns

(address)

{ return l2Gateway ; }

/* * Parse data received in outboundTransfer * @param data encoded data received * @return from account that initiated the deposit, * maxSubmissionCost max gas deducted from user's L2 balance to cover base submission fee, * extraData decoded data / function

_parseOutboundData (bytes

memory data) internal pure returns

(address

from , uint256 maxSubmissionCost , bytes

memory extraData) { // Router encoded (from , extraData)

= abi . decode (data ,

(address ,

bytes)) ;

// User encoded (maxSubmissionCost , extraData)

= abi . decode (extraData ,

(uint256 ,

bytes)) ; }

// ---------- // Custom methods // ---------- /* Disables the ability to deposit funds/ function

disableDeposits ( )

external onlyOwner { allowsDeposits =

false ; }

/* * Enables the ability to deposit funds/ function

enableDeposits ( )

external onlyOwner { require (l1CustomToken !=

address (0) ,

"Token bridge information has not been set yet") ; allowsDeposits =

true ; } } IL1CustomGateway is an interface very similar toICustomGateway , andL1CrosschainMessenger implements a method to send the cross-chain message to L2 through the Inbox.

/* * @title Minimum expected implementation of a crosschain messenger contract to be deployed on L1/ abstract contract

L1CrosschainMessenger

{ IInbox public immutable inbox ;

/* * Emitted when calling sendTxToL2CustomRefund * @param from account that submitted the retryable ticket * @param to account recipient of the retryable ticket * @param seqNum id for the retryable ticket * @param data data of the retryable ticket / event

TxToL2 (address

indexed

from , address

indexed to , uint256

indexed seqNum , bytes data) ;

constructor (address inbox_)

{ inbox =

IInbox (inbox_) ; }

modifier

onlyCounterpartGateway (address l2Counterpart)

{ // A message coming from the counterpart gateway was executed by the bridge IBridge bridge = inbox . bridge ( ) ; require (msg . sender

==

address (bridge) ,

"NOT_FROM_BRIDGE") ;

// And the outbox reports that the L2 address of the sender is the counterpart gateway address l2ToL1Sender =

IOutbox (bridge . activeOutbox ( )) . l2ToL1Sender ( ) ; require (l2ToL1Sender == l2Counterpart ,

"ONLY_COUNTERPART_GATEWAY") ;

_ ; }

/* * *Creates the retryable ticket to send over to L2 through the Inbox * @param to account to be credited with the tokens in the destination layer * @param refundTo account, or its L2 alias if it have code in L1, to be credited with excess gas refund in L2 * @param user account with rights to cancel the retryable and receive call value refund * @param l1CallValue callvalue sent in the L1 submission transaction * @param l2CallValue callvalue for the L2 message * @param maxSubmissionCost max gas deducted from user's L2 balance to cover base submission fee * @param maxGas max gas deducted from user's L2 balance to cover L2 execution * @param gasPriceBid gas price for L2 execution * @param data encoded data for the retryable * @return seqnum id for the retryable ticket* / function

_sendTxToL2CustomRefund (address to , address refundTo , address user , uint256 l1CallValue , uint256 l2CallValue , uint256 maxSubmissionCost , uint256 maxGas , uint256 gasPriceBid , bytes

memory data)

internal

returns

(uint256)

{ uint256 seqNum = inbox . createRetryableTicket { value : l1CallValue } (to , l2CallValue , maxSubmissionCost , refundTo , user , maxGas , gasPriceBid , data) ;

emit

TxToL2 (user , to , seqNum , data) ; return seqNum ; } } We now deploy that gateway to L1.

const

{ ethers }

=

require ('hardhat') ; const

{ providers ,

Wallet ,

BigNumber

}

=

require ('ethers') ; const

{ getL2Network ,

L1ToL2MessageStatus

}

=

require ('@arbitrum/sdk') ; const

{ AdminErc20Bridger , Erc20Bridger , }

=

require ('@arbitrum/sdk/dist/lib/assetBridger/erc20Bridger') ; require ('dotenv') . config ( ) ;

/* * *Set up: instantiate L1 / L2 wallets connected to providers*/ const walletPrivateKey = process . env . DEVNET_PRIVKEY ; const l1Provider =

new

providers . JsonRpcProvider (process . env . L1RPC) ; const l2Provider =

new

providers . JsonRpcProvider (process . env . L2RPC) ; const l1Wallet =

new

Wallet (walletPrivateKey , l1Provider) ; const l2Wallet =

new

Wallet (walletPrivateKey , l2Provider) ;

const

main

=

async

( )

=>

{ /* * Use l2Network to create an Arbitrum SDK AdminErc20Bridger instance * We'll use AdminErc20Bridger for its convenience methods around registering tokens to a custom gateway / const l2Network =

await

getL2Network (l2Provider) ; const erc20Bridger =

new

Erc20Bridger (l2Network) ; const adminTokenBridger =

new

AdminErc20Bridger (l2Network) ; const l1Router = l2Network . tokenBridge . l1GatewayRouter ; const l2Router = l2Network . tokenBridge . l2GatewayRouter ; const inbox = l2Network . ethBridge . inbox ;

/* * Deploy our custom gateway to L1/ const

L1CustomGateway

=

await

await ethers . getContractFactory ('L1CustomGateway' , l1Wallet) ; console . log ('Deploying custom gateway to L1') ; const l1CustomGateway =

await

L1CustomGateway . deploy (l1Router , inbox) ; await l1CustomGateway . deployed ( ) ; console . log(`custom gateway is deployed to L1 at { l1CustomGateway . address }) ; const l1CustomGatewayAddress = l1CustomGateway . address ; } ;

main ( ) . then (( )

=> process . exit (0)) . catch ((error)

=>

{ console . error (error) ; process . exit (1) ; }) ;

Step 2: Create a gateway and deploy it on L2

We'll now create the counterpart of the gateway we created on L1 and deploy it on L2. A good example of a custom gateway on L2 isArbitrum's generic-custom gateway on L2 .

As we did with the L1 gateway, we'll use a simpler approach with the same characteristics of that in L1: supports only one token and has the ability to be disabled/enabled by the owner of the contract. It will also have asetTokenBridgeInformation to be called by the owner of the contract to initialize the gateway.

// SPDX-License-Identifier: Apache-2.0 pragma

solidity

^ 0.8.0 ;

import

"./interfaces/ICustomGateway.sol" ; import

"./CrosschainMessenger.sol" ; import

"./interfaces/IArbToken.sol" ; import

"@openzeppelin/contracts/access/Ownable.sol" ;

/* * @title Example implementation of a custom gateway to be deployed on L2 * @dev Inheritance of Ownable is optional. In this case we use it to call the function setTokenBridgeInformation * and simplify the test / contract

L2CustomGateway

is IL2CustomGateway , L2CrosschainMessenger , Ownable { // Exit number (used for tradeable exits) uint256

public exitNum ;

// Token bridge state variables address

public l1CustomToken ; address

public l2CustomToken ; address

public l1Gateway ; address

public router ;

// Custom functionality bool

public allowsWithdrawals ;

/* * Contract constructor, sets the L2 router to be used in the contract's functions * @param router_ L2GatewayRouter address$ constructor (address router_)

{ router = router_ ; allowsWithdrawals =

false ; }

/* * Sets the information needed to use the gateway. To simplify the process of testing, this function can be called once * by the owner of the contract to set these addresses. * @param l1CustomToken_ address of the custom token on L1 * @param l2CustomToken_ address of the custom token on L2 * @param l1Gateway_ address of the counterpart gateway (on L1) / function

setTokenBridgeInformation (address l1CustomToken_ , address l2CustomToken_ , address l1Gateway_)

public onlyOwner { require (l1CustomToken ==

address (0) ,

"Token bridge information already set") ; l1CustomToken = l1CustomToken_ ; l2CustomToken = l2CustomToken_ ; l1Gateway = l1Gateway_ ;

// Allows withdrawals after the information has been set allowsWithdrawals =

true ; }

/// @dev See {ICustomGateway-outboundTransfer} function

outboundTransfer (address l1Token , address to , uint256 amount , bytes

calldata data)

public

payable

returns

(bytes

memory)

{ return

outboundTransfer (l1Token , to , amount ,

0 ,

0 , data) ; }

/// @dev See {ICustomGateway-outboundTransfer} function

outboundTransfer (address l1Token , address to , uint256 amount , uint256 ,

/ *_maxGas* / uint256 ,

/ *_gasPriceBid* / bytes

calldata data)

public

payable override returns

(bytes

memory res)

{ // Only execute if deposits are allowed require (allowsWithdrawals ==

true ,

"Withdrawals are currently disabled") ;

// The function is marked as payable to conform to the inheritance setup // This particular code path shouldn't have a msg.value > 0 require (msg . value ==

0 ,

"NO_VALUE") ;

// Only allow the custom token to be bridged through this gateway require (l1Token == l1CustomToken ,

"Token is not allowed through this gateway") ;

(address

from ,

bytes

memory extraData)

=

_parseOutboundData (data) ;

// The inboundEscrowAndCall functionality has been disabled, so no data is allowed require (extraData . length ==

0 ,

"EXTRA_DATA_DISABLED") ;

// Burns L2 tokens in order to release escrowed L1 tokens IArbToken (l2CustomToken) . bridgeBurn (from , amount) ;

// Current exit number for this operation uint256 currExitNum = exitNum ++ ;

// We override the res field to save on the stack res =

getOutboundCalldata (l1Token ,

from , to , amount , extraData) ;

// Trigger the crosschain message uint256 id =

_sendTxToL1 (from , l1Gateway , res) ;

emit

WithdrawalInitiated (l1Token ,

from , to , id , currExitNum , amount) ; return abi . encode (id) ; }

/// @dev See {ICustomGateway-finalizeInboundTransfer} function

finalizeInboundTransfer (address l1Token , address

from , address to , uint256 amount , bytes

calldata data)

public

payable override onlyCounterpartGateway (l1Gateway)

{ // Only allow the custom token to be bridged through this gateway require (l1Token == l1CustomToken ,

"Token is not allowed through this gateway") ;

// Abi decode may revert, but the encoding is done by L1 gateway, so we trust it (,

bytes

memory callHookData)

= abi . decode (data ,

(bytes ,

bytes)) ; if

(callHookData . length !=

0)

{ // callHookData should always be 0 since inboundEscrowAndCall is disabled callHookData =

bytes ("") ; }

// Mints L2 tokens IArbToken (l2CustomToken) . bridgeMint (to , amount) ;

emit

DepositFinalized (l1Token ,

from , to , amount) ; }

/// @dev See {ICustomGateway-getOutboundCalldata} function

getOutboundCalldata (address l1Token , address

from , address to , uint256 amount , bytes

memory data)

public

view override returns

(bytes

memory outboundCalldata)

{ outboundCalldata = abi . encodeWithSelector (ICustomGateway . finalizeInboundTransfer . selector , l1Token , from , to , amount , abi . encode (exitNum , data)) ;

return outboundCalldata ; }

/// @dev See {ICustomGateway-calculateL2TokenAddress} function

calculateL2TokenAddress (address l1Token)

public

view override returns

(address)

{ if

(l1Token == l1CustomToken)

{ return l2CustomToken ; }

return

address (0) ; }

/// @dev See {ICustomGateway-counterpartGateway} function

counterpartGateway ( )

public

view override returns

(address)

{ return l1Gateway ; }

/* * Parse data received in outboundTransfer * @param data encoded data received * @return from account that initiated the deposit, * extraData decoded data / function

_parseOutboundData (bytes

memory data) internal view returns

(address

from , bytes

memory extraData) { if

(msg . sender == router)

{ // Router encoded (from , extraData)

= abi . decode (data ,

(address ,

bytes)) ; }

else

{ from

= msg . sender ; extraData = data ; } }

// ---------- // Custom methods // ---------- /* Disables the ability to deposit funds/ function

disableWithdrawals ( )

external onlyOwner { allowsWithdrawals =

false ; }

/* * Enables the ability to deposit funds/ function

enableWithdrawals ( )

external onlyOwner { require (l1CustomToken !=

address (0) ,

"Token bridge information has not been set yet") ; allowsWithdrawals =

true ; } } IL2CustomGateway is also an interface very similar toICustomGateway , andL2CrosschainMessenger implements a method to send the cross-chain message to L1 through ArbSys.

/* * @title Minimum expected implementation of a crosschain messenger contract to be deployed on L2 abstract contract

L2CrosschainMessenger

{ address

internal

constant ARB_SYS_ADDRESS =

address (100) ;

/* * Emitted when calling sendTxToL1 * @param from account that submits the L2-to-L1 message * @param to account recipient of the L2-to-L1 message * @param id id for the L2-to-L1 message * @param data data of the L2-to-L1 message / event

TxToL1 (address

indexed

from , address

indexed to , uint256

indexed id , bytes data) ;

modifier

onlyCounterpartGateway (address l1Counterpart)

{ require (msg . sender == AddressAliasHelper . applyL1ToL2Alias (l1Counterpart) , "ONLY_COUNTERPART_GATEWAY") ;

_ ; }

/* * Creates an L2-to-L1 message to send over to L1 through ArbSys * @param from account that is sending funds from L2 * @param to account to be credited with the tokens in the destination layer * @param data encoded data for the L2-to-L1 message * @return id id for the L2-to-L1 message / function

_sendTxToL1 (address

from , address to , bytes

memory data)

internal

returns

(uint256)

{ uint256 id =

ArbSys (ARB_SYS_ADDRESS) . sendTxToL1 (to , data) ;

emit

TxToL1 (from , to , id , data) ; return id ; } } We now deploy that gateway to L2.

const

{ ethers }

=

require ('hardhat') ; const

{ providers ,

Wallet ,

BigNumber

}

=

require ('ethers') ; const

{ getL2Network ,

L1ToL2MessageStatus

}

=

require ('@arbitrum/sdk') ; const

{ AdminErc20Bridger , Erc20Bridger , }

=

require ('@arbitrum/sdk/dist/lib/assetBridger/erc20Bridger') ; require ('dotenv') . config ( ) ;

/* * Set up: instantiate L1 / L2 wallets connected to providers/ const walletPrivateKey = process . env . DEVNET_PRIVKEY ; const l1Provider =

new

providers . JsonRpcProvider (process . env . L1RPC) ; const l2Provider =

new

providers . JsonRpcProvider (process . env . L2RPC) ; const l1Wallet =

new

Wallet (walletPrivateKey , l1Provider) ; const l2Wallet =

new

Wallet (walletPrivateKey , l2Provider) ;

const

main

=

async

( )

=>

{ /* * Use l2Network to create an Arbitrum SDK AdminErc20Bridger instance * We'll use AdminErc20Bridger for its convenience methods around registering tokens to a custom gateway */ const l2Network =

await

getL2Network (l2Provider) ; const erc20Bridger =

new

Erc20Bridger (l2Network) ; const adminTokenBridger =

new

AdminErc20Bridger (l2Network) ; const l1Router = l2Network . tokenBridge . l1GatewayRouter ; const l2Router = l2Network . tokenBridge . l2GatewayRouter ; const inbox = l2Network . ethBridge . inbox ;

/* * Deploy our custom gateway to L2*/ const

L2CustomGateway

=

await

await ethers . getContractFactory ('L2CustomGateway' , l2Wallet) ; console . log ('Deploying custom gateway to L2') ; const l2CustomGateway =

await

L2CustomGateway . deploy (l2Router) ; await l2CustomGateway . deployed ( ) ; console . log (`Custom gateway is deployed to L2 at ${ l2CustomGateway . address }`) ; const l2CustomGatewayAddress = l2CustomGateway . address ; } ;

main ( ) . then (( )

=> process . exit (0)) . catch ((error)

=>

{ console . error (error) ; process . exit (1) ; }) ;

Step 3: Deploy the custom tokens on L1 and L2

This step will depend on your setup. In this case, as our simplified gateway supports only one token, we will deploy those on L1 and L2 to be able to call thesetTokenBridgeInformation method on both gateways afterwards.

We won't go through the process of deploying custom tokens in this How-to, but you can see a detailed explanation of the steps to take in the pageHow to bridge tokens via Arbitrum's generic-custom gateway

Step 4: Configure your custom tokens on your gateways

This step will also depend on your setup. In this case, our simplified gateway requires methodsetTokenBridgeInformation to be called on both gateways to set the addresses of the counterpart gateway and both custom tokens.

/* * Set the token bridge information on the custom gateways * (This is an optional step that depends on your configuration. In this example, we've added one-shot * functions on the custom gateways to set the token bridge addresses in a second step. This could be * avoided if you are using proxies or the opcode CREATE2 for example) / console . log ('Setting token bridge information on L1CustomGateway:') ; const setTokenBridgeInfoOnL1 =

await l1CustomGateway . setTokenBridgeInformation (l1CustomToken . address , l2CustomToken . address , l2CustomGatewayAddress ,) ;

const setTokenBridgeInfoOnL1Rec =

await setTokenBridgeInfoOnL1 . wait ( ) ; console . log (`Token bridge information set on L1CustomGateway! L1 receipt is: ${ setTokenBridgeInfoOnL1Rec . transactionHash }` ,) ;

console . log ('Setting token bridge information on L2CustomGateway:') ; const setTokenBridgeInfoOnL2 =

await l2CustomGateway . setTokenBridgeInformation (l1CustomToken . address , l2CustomToken . address , l1CustomGatewayAddress ,

) ;

const setTokenBridgeInfoOnL2Rec =

await setTokenBridgeInfoOnL2 . wait ( ) ; console . log (Token bridge information set on L2CustomGateway! L2 receipt is: { setTokenBridgeInfoOnL2Rec . transactionHash } ,) ;

Step 5: Register the custom token to your custom gateway

Once all contracts are deployed on their respective chains, and they all have the information of the gateways and tokens, it's time to register the token in your custom gateway.

As mentioned inHow to bridge tokens via Arbitrum's generic-custom gateway, this action needs to be done by the L1 token, and we've implemented the functionregisterTokenOnL2 to do it. So now we only need to call that function.

In this case, when using this function only one action will be performed:

1. Call functionsetGateway
2. ofL1GatewayRouter
3. . This will change thel1TokenToGateway
4. internal mapping it holds and will send a retryable ticket to the counterpartL2GatewayRouter
5. contract in L2, to also set its mapping to the new values.

To simplify the process, we'll use Arbitrum's SDK and call the methodregisterCustomToken of theAdminErc20Bridger class, which will call the registerTokenOnL2 method of the token passed by parameter.

/* * *Register the custom gateway as the gateway of our custom token*/ console . log ('Registering custom token on L2:') ; const registerTokenTx =

await adminTokenBridger . registerCustomToken (l1CustomToken . address , l2CustomToken . address , l1Wallet , l2Provider ,) ;

const registerTokenRec =

await registerTokenTx . wait ( ) ; console . log (Registering token txn confirmed on L1! L1 receipt is: { registerTokenRec . transactionHash }) ; console . log (` Waiting for L2 retryable (takes 10-15 minutes); current time: { new

Date ( ) . toTimeString ( ) }) ` ,) ;

/* * *The L1 side is confirmed; now we listen and wait for the L2 side to be executed; we can do this by computing the expected txn hash of the L2 transaction. * To compute this txn hash, we need our message's "sequence numbers", unique identifiers of each L1 to L2 message. * We'll fetch them from the event logs with a helper method.* / const l1ToL2Msgs =

await registerTokenRec . getL1ToL2Messages (l2Provider) ;

/* * *In this case, the registerTokenOnL2 method creates 1 L1-to-L2 messages to set the L1 token to the Custom Gateway via the Router * Here, We check if that message is redeemed on L2* / expect (l1ToL2Msgs . length ,

'Should be 1 message.') . to . eq (1) ;

const setGateways =

await l1ToL2Msgs [0] . waitForStatus ( ) ; expect (setGateways . status ,

'Set gateways not redeemed.') . to . eq (L1ToL2MessageStatus . REDEEMED) ;

console . log ('Your custom token and gateways are now registered on the token bridge !') ;

Conclusion

Once this step is done, your L1 and L2 gateways will be registered in the token bridge and both tokens will be connected through your custom gateway.

You can bridge tokens between L1 and L2 using the custom tokens, along with the router and gateway contracts from each layer.

If you want to see an example of bridging a token from L1 to L2 using Arbitrum's SDK, you can check outHow to bridge tokens via Arbitrum's standard ERC20 gateway , where the process is described in steps 2-5.

The full code of this how-to and a more extensive deployment and testing script can be foundin this package of our tutorials repository.

Resources

1. Concept page: Token Bridge
2. Arbitrum SDK
3. Token bridge contract addresses Edit this page Last updatedonMar 22, 2024 Previous Bridge tokens via Arbitrum's generic-custom gateway Next Bridging ether