Bridge tokens via Arbitrum’s generic-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. 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), usingArbitrum’s generic-custom gateway . For alternative ways of bridging tokens, don’t forget to check out this overview .

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 created this script in our tutorials repository that encapsulates the entire process.

Step 0: Review the prerequisites

As stated in thetoken bridge conceptual page , there are a few prerequisites to keep in mind while using this method to make a token bridgeable.

First of all, theL1 counterpart of the token , must conform to theICustomToken interface. This means that:

- It must have aisArbitrumEnabled
- method that returns0xb1
- It must have a method that makes an external call toL1CustomGateway.registerCustomL2Token
- specifying the address of the L2 contract, and toL1GatewayRouter.setGateway
- specifying the address of the custom gateway. These calls should be made only once to configure the gateway.

These methods are needed to register the token via the gateway contract. If your L1 contract does not include these methods and it is not upgradeable, registration could alternatively be performed in one of these ways:

- As a chain-owner registration via anArbitrum DAO
- proposal.
- By wrapping your L1 token and registering the wrapped version of your token.

Keep in mind that this registration can only be done once.

Also, theL2 counterpart of the token , must conform to theIArbToken interface. This means that:

- It must havebridgeMint
- andbridgeBurn
- methods only callable by the L2CustomGateway contract
- It must have anl1Address
- view method that returns the address of the token in L1

Token compatibility with available tooling If you want your token to be compatible out of the box with all the tooling available (e.g., theArbitrum bridge), we recommend that you keep the implementation of the IArbToken interface as close as possible to theL2GatewayToken implementation example.

For example, if an allowance check is added to thebridgeBurn() function, the token will not be easily withdrawable through the Arbitrum bridge UI, as the UI does not prompt an approval transaction of tokens by default (it expects the tokens to follow the recommended L2GatewayToken implementation).

Step 1: Create a token and deploy it on L1

We‘ll begin the process by creating and deploying on L1 a sample token to bridge. If you already have a token contract on L1, you don’t need to perform this step.

However, you will need to upgrade the contract if it doesn’t include the required methods described in the previous step.

We first create a standard ERC20 contract using OpenZeppelin’s implementation. We make only 1 adjustment to that implementation, for simplicity, although it is not required: we specify aninitialSupply to be pre-minted and sent to the deployer address upon creation.

We’ll also add the required methods to make our token bridgeable via the generic-custom gateway.

// SPDX-License-Identifier: MIT pragma

solidity

^ 0.8.0 ;

import

"./interfaces/ICustomToken.sol" ; import

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

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

/* * *@title Interface needed to call function registerTokenToL2 of the L1CustomGateway*/ interface

IL1CustomGateway

{ function

registerTokenToL2 (address _l2Address , uint256 _maxGas , uint256 _gasPriceBid , uint256 _maxSubmissionCost , address _creditBackAddress)

external

payable

returns

(uint256) ; }

/* * *@title Interface needed to call function setGateway of the L2GatewayRouter*/ interface

IL2GatewayRouter

{ function

setGateway (address _gateway , uint256 _maxGas , uint256 _gasPriceBid , uint256 _maxSubmissionCost , address _creditBackAddress)

external

payable

returns

(uint256) ; }

contract

L1Token

is Ownable , ICustomToken , ERC20 { address

private customGatewayAddress ; address

private routerAddress ; bool

private shouldRegisterGateway ;

/* * *@dev See {ERC20-constructor} and {Ownable-constructor} * * An initial supply amount is passed, which is preminted to the deployer.* / constructor (address _customGatewayAddress ,

address _routerAddress ,

uint256 _initialSupply)

ERC20 ("L1CustomToken" ,

"LCT")

{ customGatewayAddress = _customGatewayAddress ; routerAddress = _routerAddress ; _mint (msg . sender , _initialSupply *

10

**

decimals ( )) ; }

/// @dev we only set shouldRegisterGateway to true when in registerTokenOnL2 function

isArbitrumEnabled ( )

external

view override returns

(uint8)

{ require (shouldRegisterGateway ,

"NOT_EXPECTED_CALL") ; return

uint8 (0xb1) ; }

/// @dev See {ICustomToken-registerTokenOnL2} function

registerTokenOnL2 (address l2CustomTokenAddress , uint256 maxSubmissionCostForCustomGateway , uint256 maxSubmissionCostForRouter , uint256 maxGasForCustomGateway , uint256 maxGasForRouter , uint256 gasPriceBid , uint256 valueForGateway , uint256 valueForRouter , address creditBackAddress)

public override payable onlyOwner { // we temporarily setshouldRegisterGateway to true for the callback in registerTokenToL2 to succeed bool prev = shouldRegisterGateway ; shouldRegisterGateway =

true ;

IL1CustomGateway (customGatewayAddress) . registerTokenToL2 { value : valueForGateway } (l2CustomTokenAddress , maxGasForCustomGateway , gasPriceBid , maxSubmissionCostForCustomGateway , creditBackAddress) ;

IL2GatewayRouter (routerAddress) . setGateway { value : valueForRouter } (customGatewayAddress , maxGasForRouter , gasPriceBid , maxSubmissionCostForRouter , creditBackAddress) ;

shouldRegisterGateway

prev ; }

/// @dev See {ERC20-transferFrom} function

transferFrom (address sender , address recipient , uint256 amount)

public

override (ICustomToken , ERC20)

returns

(bool)

{ return super . transferFrom (sender , recipient , amount) ; }

/// @dev See {ERC20-balanceOf} function

balanceOf (address account)

public

view

override (ICustomToken , ERC20)

returns

(uint256)

{ return super . balanceOf (account) ; } } We now deploy that token to L1.

const

{ ethers }

=

require ('hardhat') ; const

{ providers ,

Wallet

}

=

require ('ethers') ; const

{ getL2Network }

=

require ('@arbitrum/sdk') ; require ('dotenv') . config ( ) ;

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) ;

/* * *For the purpose of our tests, here we deploy an standard ERC20 token (L1Token) to L1 * It sends its deployer (us) the initial supply of 1000* / const

main

=

async

( )

=>

{ /* * *Use l2Network to get the token bridge addresses needed to deploy the token*/ const l2Network =

await

getL2Network (l2Provider) ;

const l1Gateway = l2Network . tokenBridge . l1CustomGateway ; const l1Router = l2Network . tokenBridge . l1GatewayRouter ;

/* * *Deploy our custom token smart contract to L1 * We give the custom token contract the address of l1CustomGateway and l1GatewayRouter as well as the initial supply (premine)* / console . log ('Deploying the test L1Token to L1:') ; const

L1Token

=

await

(await ethers . getContractFactory ('L1Token')) . connect (l1Wallet) ; const l1Token =

await

L1Token . deploy (l1Gateway , l1Router ,

1000) ;

await l1Token . deployed ( ) ; console . log (L1Token is deployed to L1 at { l1Token . address }) ;

/* * *Get the deployer token balance*/ const tokenBalance =

await l1Token . balanceOf (l1Wallet . address) ; console . log (nitial token balance of deployer: { tokenBalance }) ; } ;

main ( ) . then (( )

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

=>

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

Step 2: Create a token and deploy it on L2

We'll now create and deploy on L2 the counterpart of the token we created on L1.

We'll create a standard ERC20 contract using OpenZeppelin's implementation, and add the required methods fromIArbToken .

```
// SPDX-License-Identifier: Apache-2.0 pragma
solidity
^ 0.8.0 ;
import
"./interfaces/IArbToken.sol" ; import
"@openzeppelin/contracts/token/ERC20/ERC20.sol" ;
contract
L2Token
is ERC20 , IArbToken { address
public l2Gateway ; address
public override l1Address ;
modifier
onlyL2Gateway ( )
{ require ( msg . sender == l2Gateway ,
"NOT_GATEWAY" ) ; _ ; }
constructor ( address _l2Gateway ,
address _l1TokenAddress )
ERC20 ( "L2CustomToken" ,
"LCT" )
{ l2Gateway = _l2Gateway ; l1Address = _l1TokenAddress ; }
/* * @notice should increase token supply by amount, and should only be callable by the L2Gateway/ function
bridgeMint ( address account ,
uint256 amount )
external override onlyL2Gateway { _mint ( account , amount ) ; }
/* * @notice should decrease token supply by amount, and should only be callable by the L2Gateway/ function
bridgeBurn ( address account ,
uint256 amount )
external override onlyL2Gateway { _burn ( account , amount ) ; }
// Add any extra functionality you want your token to have. } We now deploy that token to L2.
const
{ ethers }
=
require ( 'hardhat' ) ; const
{ providers ,
Wallet
}
=
require ( 'ethers' ) ; const
{ getL2Network }
```

=

require ('@arbitrum/sdk') ; require ('dotenv') . config ( ) ;

const walletPrivateKey = process . env . DEVNET_PRIVKEY ; const l2Provider =

new

providers . JsonRpcProvider (process . env . L2RPC) ; const l2Wallet =

new

Wallet (walletPrivateKey , l2Provider) ;

const l1TokenAddress =

'

' ;

/* * For the purpose of our tests, here we deploy an standard ERC20 token (L2Token) to L2 const

main

=

async

( )

=>

{ /* * Use l2Network to get the token bridge addresses needed to deploy the token const l2Network =

await

getL2Network (l2Provider) ; const l2Gateway = l2Network . tokenBridge . l2CustomGateway ;

/* * Deploy our custom token smart contract to L2 * We give the custom token contract the address of l2CustomGateway as well as the address of the counterpart L1 token / console . log ('Deploying the test L2Token to L2:') ; const

L2Token

=

await

(await ethers . getContractFactory ('L2Token')) . connect (l2Wallet) ; const l2Token =

await

L2Token . deploy (l2Gateway , l1TokenAddress) ;

await l2Token . deployed ( ) ; console . log (L2Token is deployed to L2 at { l2Token . address }) ; } ;

main ( ) . then (( )

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

=>

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

Step 3: Register the custom token to the generic-custom gateway

Once both our contracts are deployed in their respective chains, it's time to register the token in the generic-custom gateway.

As mentioned before, 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.

When using this function two actions will be performed:

1. Call functionregisterTokenToL2

2. *ofL1CustomGateway*
3. *. This will change theI1ToL2Token*
4. *internal mapping it holds and will send a retryable ticket to the counterpartL2CustomGateway*
5. *contract in L2, to also set its mapping to the new values.*
6. *Call functionsetGateway*
7. *ofL1GatewayRouter*
8. *. This will change theI1TokenToGateway*
9. *internal mapping it holds and will send a retryable ticket to the counterpartL2GatewayRouter*
10. *contract in L2, to also set its mapping to the new values.*

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

/ * Register custom token on our custom gateway / const adminTokenBridger =*

new

AdminErc20Bridger (l2Network) ; 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 ,) ;

/ * 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 principle, a single L1 txn can trigger any number of L1-to-L2 messages (each with its own sequencer number). * In this case, the registerTokenOnL2 method created 2 L1-to-L2 messages; * - (1) one to set the L1 token to the Custom Gateway via the Router, and * - (2) another to set the L1 token to its L2 token address via the Generic-Custom Gateway * Here, We check if both messages are redeemed on L2 / expect (l1ToL2Msgs . length ,*

'Should be 2 messages.') . to . eq (2) ;

const setTokenTx =

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

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

const setGateways =

await l1ToL2Msgs [1] . waitForStatus ( ) ; expect (setGateways . status ,

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

console . log ('Your custom token is now registered on our custom gateway Go ahead and make the deposit!' ,) ;

Conclusion

Once this step is done, your L1 and L2 tokens will be connected through the generic-custom gateway.

You can bridge tokens between L1 and L2 using the origin L1 token and the custom token deployed on L2, 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.

Frequently asked questions

Can I run the same register token process multiple times for the same L1 token?

No, you can only register once an L2 token for the same L1 token. After that, the call toregisterTokenToL2 will revert if run again.

What can I do if my L1 token is not upgradable?

As mentioned in the concept page, the token registration can alternatively be performed as a chain-owner registration via Arbitrum DAO proposal.

Can I set up the generic-custom gateway after a standard ERC20 token exists on L2?

Yes, if your token has a standard ERC20 counterpart on L2, you can go through the process of registering your custom L2 token as outlined in this page. At that moment, your L1 token will have 2 counterpart tokens on L2, but only your new custom L2 token will be minted when depositing tokens from L1 (L1-to-L2 bridging). Both L2 tokens will be withdrawable (L2-to-L1 bridging), so users holding the old standard ERC20 token will be able to withdraw back to L1 (using the L2CustomGateway contract instead of the bridge UI) and then deposit to L2 to get the new custom L2 tokens.

Resources

1. Concept page: Token Bridge
2. Arbitrum SDK
3. Token bridge contract addresses

Edit this page Last updatedonMar 26, 2024 Previous Bridge tokens via Arbitrum's standard ERC20 gateway Next How to bridge tokens via a custom gateway