ERC-20 token bridging

The Arbitrum protocol itself technically has no native notion of any token standards, and gives no built-in advantage or special recognition to any particular token bridge. In this page we describe the "canonical bridge", which was implemented by Offchain Labs, and should be the primary bridge most users and applications use; it is (effectively) a decentralized app (dApp) with contracts on both Ethereum (the Layer 1, or L1) and Arbitrum (the Layer 2, or L2) that leverages Arbitrum'scross-chain message passing system to achieve basic desired token-bridging functionality. We recommend that you use it!

Design rationale

In our token bridge design, we use the term "gateway" as pethis proposal ; i.e., one of a pair of contracts on two different domains (i.e., Ethereum and an Arbitrum chain), used to facilitate cross-domain asset transfers.

We now describe some core goals that motivated the design of our bridging system.

Custom gateway functionality

For many ERC-20 tokens, "standard" bridging functionality is sufficient, which entails the following: a token contract on Ethereum is associated with a "paired" token contract on Arbitrum.

Depositing a token entails escrowing some amount of the token in an L1 bridge contract, and minting the same amount at the paired token contract on L2. On L2, the paired contract behaves much like a normal ERC-20 token contract. Withdrawing entails burning some amount of the token in the L2 contract, which then can later be claimed from the L1 bridge contract.

Many tokens, however, require custom gateway systems, the possibilities of which are hard to generalize, e.g.:

- Tokens which accrue interest to their holders need to ensure that the interest is dispersed properly across layers, and doesn't simply accrue to the bridge contracts
- Our cross-domain WETH implementations requires tokens be wrapped and unwrapped as they move across layers.

Thus, our bridge architecture must allow not just the standard deposit and withdraw functionalities, but for new, custom gateways to be dynamically added over time.

Canonical L2 representation per L1 token contract

Having multiple custom gateways is well and good, but we also want to avoid a situation in which a single L1 token that uses our bridging system can be represented at multiple addresses/contracts on the L2 as this adds significant friction and confusion for users and developers. Thus, we need a way to track which L1 token uses which gateway, and in turn, to have a canonical address oracle that maps the tokens addresses across the Ethereum and Arbitrum domains.

Canonical token bridge implementation

With this in mind, we provide an overview of our token bridging architecture.

Our architecture consists of three types of contracts:

1. Asset contracts
2. : These are the token contracts themselves, i.e., an ERC-20 on L1 and it's counterpart on Arbitrum.
3. Gateways
4. : Pairs of contracts (one on L1, one on L2) that implement a particular type of cross-chain asset bridging.
5. Routers
6. : Exactly two contracts (one on L1, one on L2) that route each asset to its designated gateway.

All Ethereum to Arbitrum token transfers are initiated via the router contract on L1, theL1GatewayRouter contract.L1GatewayRouter forwards the token's deposit call to the appropriate gateway contract on L1, theL1ArbitrumGateway contract.L1GatewayRouter is responsible for mapping L1 token addresses to L1Gateway contracts, thus acting as an L1/L2 address oracle, and ensuring that each token corresponds to only one gateway. TheL1ArbitrumGateway then communicates to its counterpart gateway contract on L2, theL2ArbitrumGateway contract (typically/expectedly viaretryable tickets).

Similarly, Arbitrum to Ethereum transfers are initiated via the router contract on L2, theL2GatewayRouter contract, which calls the token's gateway contract on L2, theL2ArbitrumGateway contract, which in turn communicates to its corresponding gateway contract on L1, theL1ArbitrumGateway contract (typically/expectedly viasending L2-to-L1 messages to the outbox).

For any given gateway pairing, we require that calls be initiated through the corresponding router (L1GatewayRouter orL2GatewayRouter), and that the gateways conform to theTokenGateway interfaces; theTokenGateway interfaces should be flexible and extensible enough to support any bridging functionality a particular token may require.

The standard ERC-20 gateway

By default, any ERC-20 token on L1 that isn't registered to a gateway can be permissionlessly bridged through theStandardERC20Gateway .

You can use thebridge UI or follow the instructions inHow to bridge tokens via Arbitrum's standard ERC-20 gatewayto bridge a token to L2 via this gateway.

Example: Standard Arb-ERC20 deposit and withdraw

To help illustrate what this all looks like in practice, let's go through the steps of what depositing and withdrawingSomeERC20Token via our standard ERC-20 gateway looks like. Here, we're assuming thatSomeERC20Token has already been registered in theL1GatewayRouter to use the standard ERC-20 gateway.

Deposits

1. A user callsL1GatewayRouter.outboundTransferCustomRefund
2. [1]
3. (withSomeERC20Token
4. 's L1 address as an argument).
5. L1GatewayRouter
6. looks upSomeERC20Token
7. 's gateway, and finds that it's the standard ERC-20 gateway (theL1ERC20Gateway
8. contract).
9. L1GatewayRouter
10. callsL1ERC20Gateway.outboundTransferCustomRefund
11. , forwarding the appropriate parameters.
12. L1ERC20Gateway
13. escrows the tokens sent and creates a retryable ticket to triggerL2ERC20Gateway
14. 'sfinalizeInboundTransfer
15. method on L2.
16. L2ERC20Gateway.finalizeInboundTransfer
17. mints the appropriate amount of tokens at thearbSomeERC20Token
18. contract on L2.

![1]Please keep in mind that some older custom gateways might not haveoutboundTransferCustomRefund implemented andL1GatewayRouter.outboundTransferCustomRefund does not fallback tooutboundTransfer . In those cases, please use functionL1GatewayRouter.outboundTransfer .

Note that arbSomeERC20Token is an instance ofStandardArbERC20 , which includesbridgeMint andbridgeBurn methods only callable by the L2ERC20Gateway.

Withdrawals

1. On Arbitrum, a user callsL2GatewayRouter.outBoundTransfer
2. , which in turn callsoutBoundTransfer
3. on arbSomeERC20Token's gateway (i.e., L2ERC20Gateway).
4. This burns arbSomeERC20Token tokens, and callsArbSys
5. with an encoded message toL1ERC20Gateway.finalizeInboundTransfer
6. , which will be eventually executed on L1.
7. After the dispute window expires and the assertion with the user's transaction is confirmed, a user can callOutbox.executeTransaction
8. , which in turn calls the encodedL1ERC20Gateway.finalizeInboundTransfer
9. message, releasing the user's tokens from theL1ERC20Gateway
10. contract's escrow.

The Arbitrum generic-custom gateway

Just because a token has requirements beyond what are offered via the standard ERC-20 gateway, that doesn't necessarily mean that a unique gateway needs to be tailor-made for the token in question. Our generic-custom gateway is designed to be flexible enough to be suitable for most (but not necessarily all) custom fungible token needs. As a general rule:

If your custom token has the ability to increase its supply (i.e., mint) directly on the L2, and you want the L2-minted tokens be withdrawable back to L1 and recognized by the L1 contract, it will probably require its own special gateway. Otherwise, the generic-custom gateway is likely the right solution for you!

Some examples of token features suitable for the generic-custom gateway:

- An L2 token contract upgradable via a proxy
- An L2 token contract that includes address whitelisting/blacklisting
- The deployer determines the address of the L2 token contract

Setting up your token with the generic-custom gateway

Follow the following steps to get your token set up to use the generic-custom gateway. You can also find more detailed instructions in the pageHow to bridge tokens via Arbitrum's generic-custom gateway.

1. Have an L1 token

Your token on L1 should conform to theICustomToken interface (seeTestCustomTokenL1 for an example implementation). Crucially, it must have anisArbitrumEnabled method in its interface.

1. Deploy your token on Arbitrum

Your token should conform to the minimumIArbToken interface; i.e., it should havebridgeMint andbridgeBurn methods only callable by the L2CustomGateway contract, and the address of its corresponding Ethereum token accessible vial1Address . For an example implementation, seeL2GatewayToken .

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). 2. Register your token on L1 to your token on L2 via the L1CustomGateway contract

Have your L1 token's contract make an external call toL1CustomGateway.registerTokenToL2 . This registration can alternatively be performed as a chain-owner registration via anArbitrum DAO proposal.

1. Register your token on L1 to the L1GatewayRouter

After your token's registration to the generic-custom gateway is complete, have your L1 token's contract make an external call toL1GatewayRouter.setGateway ; this registration can also alternatively be performed as a chain-owner registration via anArbitrum DAO proposal.

We are here to help If you have questions about your custom token needs, feel free to reach out on ourDiscord server .

Other flavors of gateways

Note that in the system described above, one pair of gateway contracts handles the bridging of many ERC-20s; i.e., many ERC-20s on L1 are each paired with their own ERC-20s on Arbitrum via a single gateway contract pairing. Other gateways may well bear different relations with the contracts that they bridge.

Take our wrapped Ether implementation for example: here, a single WETH contract on L1 is connected to a single WETH contract on L2. When transferring WETH from one domain to another, the L1/L2 gateway architecture is used to unwrap the WETH on domain A, transfer the now-unwrapped Ether, and then re-wrap it on domain B. This ensures that WETH can behave on Arbitrum the way users are used to it behaving on Ethereum, while ensuring that all WETH tokens are always fully collateralized on the layer in which they reside.

No matter the complexity of a particular token's bridging needs, a gateway can in principle be created to accommodate it within our canonical bridging system.

You can find an example of implementation of a custom gateway in the pageHow to bridge tokens via a custom gateway.

Demos

OurHow to bridge tokens section provides example of interacting with Arbitrum's token bridge via theArbitrum SDK .

A word of caution on bridges (aka, "I've got a bridge to sell you")

Cross chain bridging is an exciting design space; alternative bridge designs can potentially offer faster withdrawals, interoperability with other chains, different trust assumptions with their own potentially valuable UX tradeoffs, etc. They can also potentially be completely insecure and/or outright scams. Users should treat other, non-canonical bridge applications the same way they treat any application running on Arbitrum, and exercise caution and due diligence before entrusting them with their value.