---

title: "Optimism standard bridge contract walkthrough" description: How does the standard bridge for Optimism work? Why does it work this way? author: Ori Pomerantz tags: ["solidity", "bridge", "layer 2"] skill: intermediate published: 2022-03-30 lang: en

---

Optimism is an Optimistic Rollup. Optimistic rollups can process transactions for a much lower price than Ethereum Mainnet (also known as layer 1 or L1) because transactions are only processed by a few nodes, instead of every node on the network. At the same time, the data is all written to L1 so everything can be proved and reconstructed with all the integrity and availability guarantees of Mainnet.

To use L1 assets on Optimism (or any other L2), the assets need to bebridged. One way to achieve this is for users to lock assets (ETH and ERC-20 tokens are the most common ones) on L1, and receive equivalent assets to use on L2. Eventually, whoever ends up with them might want to bridge them back to L1. When doing this, the assets are burned on L2 and then released back to the user on L1.

This is the way the Optimism standard bridge works. In this article we go over the source code for that bridge to see how it works and study it as an example of well written Solidity code.

Control flows {#control-flows}

The bridge has two main flows:

- Deposit (from L1 to L2)
- Withdrawal (from L2 to L1)

Deposit flow {#deposit-flow}

Layer 1 {#deposit-flow-layer-1}

1. If depositing an ERC-20, the depositor gives the bridge an allowance to spend the amount being deposited
2. The depositor calls the L1 bridge (`depositERC20`, `depositERC20To`, `depositETH`, or `depositETHTo`)
3. The L1 bridge takes possession of the bridged asset
4. ETH: The asset is transferred by the depositor as part of the call
5. ERC-20: The asset is transferred by the bridge to itself using the allowance provided by the depositor
6. The L1 bridge uses the cross-domain message mechanism to call `finalizeDeposit` on the L2 bridge

Layer 2 {#deposit-flow-layer-2}

1. The L2 bridge verifies the call to `finalizeDeposit` is legitimate:
2. Came from the cross domain message contract
3. Was originally from the bridge on L1
4. The L2 bridge checks if the ERC-20 token contract on L2 is the correct one:
5. The L2 contract reports that its L1 counterpart is the same as the one the tokens came from on L1
6. The L2 contract reports that it supports the correct interface (using ERC-165).
7. If the L2 contract is the correct one, call it to mint the appropriate number of tokens to the appropriate address. If not, start a withdrawal process to allow the user to claim the tokens on L1.

Withdrawal flow {#withdrawal-flow}

Layer 2 {#withdrawal-flow-layer-2}

1. The withdrawer calls the L2 bridge (`withdraw` or `withdrawTo`)
2. The L2 bridge burns the appropriate number of tokens belonging to `msg.sender`
3. The L2 bridge uses the cross-domain message mechanism to call `finalizeETHWithdrawal` or `finalizeERC20Withdrawal` on the L1 bridge

Layer 1 {#withdrawal-flow-layer-1}

1. The L1 bridge verifies the call to `finalizeETHWithdrawal` or `finalizeERC20Withdrawal` is legitimate:

2. Came from the cross domain message mechanism
3. Was originally from the bridge on L2
4. The L1 bridge transfers the appropriate asset (ETH or ERC-20) to the appropriate address

Layer 1 code {#layer-1-code}

This is the code that runs on L1, the Ethereum Mainnet.

IL1ERC20Bridge {#IL1ERC20Bridge}

This interface is defined here. It includes functions and definitions required for bridging ERC-20 tokens.

```
solidity // SPDX-License-Identifier: MIT
```

Most of Optimism's code is released under the MIT license.

```
solidity pragma solidity >0.5.0 <0.9.0;
```

At writing the latest version of Solidity is 0.8.12. Until version 0.9.0 is released, we don't know if this code is compatible with it or not.

```solidity / * @title IL1ERC20Bridge / interface IL1ERC20Bridge { /* * Events * ***/

event ERC20DepositInitiated(

```

In Optimism bridge terminology *deposit* means transfer from L1 to L2, and *withdrawal* means a transfer from L2 to L1.

```
solidity address indexed _l1Token, address indexed _l2Token,
```

In most cases the address of an ERC-20 on L1 is not the same the address of the equivalent ERC-20 on L2. You can see the list of token addresses here. The address with `chainId` 1 is on L1 (Mainnet) and the address with `chainId` 10 is on L2 (Optimism). The other two `chainId` values are for the Kovan test network (42) and the Optimistic Kovan test network (69).

```
solidity address indexed _from, address _to, uint256 _amount, bytes _data );
```

It is possible to add notes to transfers, in which case they are added to the events that report them.

```
solidity event ERC20WithdrawalFinalized( address indexed _l1Token, address indexed _l2Token, address indexed _from,
address _to, uint256 _amount, bytes _data );
```

The same bridge contract handles transfers in both directions. In the case of the L1 bridge, this means initialization of deposits and finalization of withdrawals.

```solidity

/********************
 * Public Functions *
 ********************/

/**
 * @dev get the address of the corresponding L2 bridge contract.
 * @return Address of the corresponding L2 bridge contract.
 */
function l2TokenBridge() external returns (address);

```

This function is not really needed, because on L2 it is a predeployed contract, so it is always at address `0x4200000000000000000000000000000000000010`. It is here for symmetry with the L2 bridge, because the address of the L1 bridge is *not* trivial to know.

```
solidity /** * @dev deposit an amount of the ERC20 to the caller's balance on L2. * @param _l1Token Address of the
L1 ERC20 we are depositing * @param _l2Token Address of the L1 respective L2 ERC20 * @param _amount Amount of the
ERC20 to deposit * @param _l2Gas Gas limit required to complete the deposit on L2. * @param _data Optional data to
forward to L2. This data is provided * solely as a convenience for external contracts. Aside from enforcing a
maximum * length, these contracts provide no guarantees about its content. */ function depositERC20( address
_l1Token, address _l2Token, uint256 _amount, uint32 _l2Gas, bytes calldata _data ) external;
```

The `_l2Gas` parameter is the amount of L2 gas the transaction is allowed to spend. Up to a certain (high) limit, this is free, so unless the ERC-20 contract does something really strange when minting, it should not be an issue. This function takes care of the common scenario, where a user bridges assets to the same address on a different blockchain.

```
solidity /** * @dev deposit an amount of ERC20 to a recipient's balance on L2. * @param _l1Token Address of the L1 ERC20 we are depositing * @param _l2Token Address of the L1 respective L2 ERC20 * @param _to L2 address to credit the withdrawal to. * @param _amount Amount of the ERC20 to deposit. * @param _l2Gas Gas limit required to complete the deposit on L2. * @param _data Optional data to forward to L2. This data is provided * solely as a convenience for external contracts. Aside from enforcing a maximum * length, these contracts provide no guarantees about its content. */ function depositERC20To( address _l1Token, address _l2Token, address _to, uint256 _amount, uint32 _l2Gas, bytes calldata _data ) external;
```

This function is almost identical to `depositERC20`, but it lets you send the ERC-20 to a different address.

```solidity /**** * Cross-chain Functions * ****/

/**
 * @dev Complete a withdrawal from L2 to L1, and credit funds to the recipient's balance of the
 * L1 ERC20 token.
 * This call will fail if the initialized withdrawal from L2 has not been finalized.
 *
 * @param _l1Token Address of L1 token to finalizeWithdrawal for.
 * @param _l2Token Address of L2 token where withdrawal was initiated.
 * @param _from L2 address initiating the transfer.
 * @param _to L1 address to credit the withdrawal to.
 * @param _amount Amount of the ERC20 to deposit.
 * @param _data Data provided by the sender on L2. This data is provided
 *   solely as a convenience for external contracts. Aside from enforcing a maximum
 *   length, these contracts provide no guarantees about its content.
 */
function finalizeERC20Withdrawal(
    address _l1Token,
    address _l2Token,
    address _from,
    address _to,
    uint256 _amount,
    bytes calldata _data
) external;

}```
```

Withdrawals (and other messages from L2 to L1) in Optimism are a two step process:

1. An initiating transaction on L2.
2. A finalizing or claiming transaction on L1. This transaction needs to happen after the fault challenge period for the L2 transaction ends.

IL1StandardBridge {#il1standardbridge}

This interface is defined here. This file contains event and function definitions for ETH. These definitions are very similar to those defined in `IL1ERC20Bridge` above for ERC-20.

The bridge interface is divided between two files because some ERC-20 tokens require custom processing and cannot be handled by the standard bridge. This way the custom bridge that handles such a token can implement `IL1ERC20Bridge` and not have to also bridge ETH.

```solidity // SPDX-License-Identifier: MIT pragma solidity >0.5.0 <0.9.0;

import "./IL1ERC20Bridge.sol";

/ * @title IL1StandardBridge / interface IL1StandardBridge is IL1ERC20Bridge { /* * Events * ****/ event ETHDepositInitiated( address indexed _from, address indexed _to, uint256 _amount, bytes _data ); ```

This event is nearly identical to the ERC-20 version (`ERC20DepositInitiated`), except without the L1 and L2 token addresses. The same is true for the other events and the functions.

```solidity event ETHWithdrawalFinalized( . . . );

/********************
 * Public Functions *
 ********************/
```
```

```
/**
 * @dev Deposit an amount of the ETH to the caller's balance on L2.
        .
        .
        .
 */
function depositETH(uint32 _l2Gas, bytes calldata _data) external payable;

/**
 * @dev Deposit an amount of ETH to a recipient's balance on L2.
        .
        .
        .
 */
function depositETHTo(
    address _to,
    uint32 _l2Gas,
    bytes calldata _data
) external payable;

/************************
 * Cross-chain Functions *
 ************************/

/**
 * @dev Complete a withdrawal from L2 to L1, and credit funds to the recipient's balance of the
 * L1 ETH token. Since only the xDomainMessenger can call this function, it will never be called
 * before the withdrawal is finalized.
            .
            .
            .
 */
function finalizeETHWithdrawal(
    address _from,
    address _to,
    uint256 _amount,
    bytes calldata _data
) external;

}
```

CrossDomainEnabled {#crossdomainenabled}

This contract is inherited by both bridges (L1 and L2) to send messages to the other layer.

```solidity // SPDX-License-Identifier: MIT pragma solidity >0.5.0 <0.9.0;

/ *Interface Imports* / import { ICrossDomainMessenger } from "./ICrossDomainMessenger.sol"; ```

This interface tells the contract how to send messages to the other layer, using the cross domain messenger. This cross domain messenger is a whole other system, and deserves its own article, which I hope to write in the future.

```solidity / **\* @title CrossDomainEnabled \* @dev Helper contract for contracts performing cross-domain communications \* \* Compiler used: defined by inheriting contract** / ***contract CrossDomainEnabled { /\* \* Variables \* \****/

```
// Messenger contract used to send and receive messages from the other domain.
address public messenger;

/***************
 * Constructor *
 ***************/

/**
 * @param _messenger Address of the CrossDomainMessenger on the current layer.
 */
constructor(address _messenger) {
    messenger = _messenger;
}
```

```

The one parameter that the contract needs to know, the address of the cross domain messenger on this layer. This parameter is set once, in the constructor, and never changes.

```solidity
```
/**********************
```

```
 * Function Modifiers *
 **********************/

/**
 * Enforces that the modified function is only callable by a specific cross-domain account.
 * @param _sourceDomainAccount The only account on the originating domain which is
 *  authenticated to call this function.
 */
modifier onlyFromCrossDomainAccount(address _sourceDomainAccount) {

```

The cross domain messaging is accessible by any contract on the blockchain where it is running (either Ethereum mainnet or Optimism). But we need the bridge on each side to *only* trust certain messages if they come from the bridge on the other side.

```
solidity require( msg.sender == address(getCrossDomainMessenger()), "OVM_XCHAIN: messenger contract unauthenticated"
);
```

Only messages from the appropriate cross domain messenger (`messenger`, as you see below) can be trusted.

```solidity
    require(
        getCrossDomainMessenger().xDomainMessageSender() == _sourceDomainAccount,
        "OVM_XCHAIN: wrong sender of cross-domain message"
    );

```

The way the cross domain messenger provides the address that sent a message with the other layer is [the `.xDomainMessageSender()` function](). As long as it is called in the transaction that was initiated by the message it can provide this information.

We need to make sure that the message we received came from the other bridge.

```solidity
    _;
}

/**********************
 * Internal Functions *
 **********************/

/**
 * Gets the messenger, usually from storage. This function is exposed in case a child contract
 * needs to override.
 * @return The address of the cross-domain messenger contract which should be used.
 */
function getCrossDomainMessenger() internal virtual returns (ICrossDomainMessenger) {
    return ICrossDomainMessenger(messenger);
}

```

This function returns the cross domain messenger. We use a function rather than the variable `messenger` to allow contracts that inherit from this one to use an algorithm to specify which cross domain messenger to use.

```solidity
/**
 * Sends a message to an account on another domain
 * @param _crossDomainTarget The intended recipient on the destination domain
 * @param _message The data to send to the target (usually calldata to a function with
 *  `onlyFromCrossDomainAccount()`)
 * @param _gasLimit The gasLimit for the receipt of the message on the target domain.
 */
function sendCrossDomainMessage(
    address _crossDomainTarget,
    uint32 _gasLimit,
    bytes memory _message

```

Finally, the function that sends a message to the other layer.

```
solidity ) internal { // slither-disable-next-line reentrancy-events, reentrancy-benign
```

Slither is a static analyzer Optimism runs on every contract to look for vulnerabilities and other potential problems. In this case, the following line triggers two vulnerabilities:

1. Reentrancy events
2. Benign reentrancy

```
solidity getCrossDomainMessenger().sendMessage(_crossDomainTarget, _message, _gasLimit); } }
```

In this case we are not worried about reentrancy we know `getCrossDomainMessenger()` returns a trustworthy address, even if Slither has no way to know that.

The L1 bridge contract {#the-l1-bridge-contract}

The source code for this contract is here.

```
solidity // SPDX-License-Identifier: MIT pragma solidity ^0.8.9;
```

The interfaces can be part of other contracts, so they have to support a wide range of Solidity versions. But the bridge itself is our contract, and we can be strict about what Solidity version it uses.

```
solidity /* Interface Imports */ import { IL1StandardBridge } from "./IL1StandardBridge.sol"; import {
IL1ERC20Bridge } from "./IL1ERC20Bridge.sol";
```

IL1ERC20Bridge and IL1StandardBridge are explained above.

```
solidity import { IL2ERC20Bridge } from "../../L2/messaging/IL2ERC20Bridge.sol";
```

This interface lets us create messages to control the standard bridge on L2.

```
solidity import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
```

This interface lets us control ERC-20 contracts. You can read more about it here.

```
solidity /* Library Imports */ import { CrossDomainEnabled } from "../../libraries/bridge/CrossDomainEnabled.sol";
```

As explained above, this contract is used for interlayer messaging.

```
solidity import { Lib_PredeployAddresses } from "../../libraries/constants/Lib_PredeployAddresses.sol";
```

`Lib_PredeployAddresses` has the addresses for the L2 contracts that always have the same address. This includes the standard bridge on L2.

```
solidity import { Address } from "@openzeppelin/contracts/utils/Address.sol";
```

OpenZeppelin's Address utilities. It is used to distinguish between contract addresses and those belonging to externally owned accounts (EOA).

Note that this isn't a perfect solution, because there is no way to distinguish between direct calls and calls made from a contract's constructor, but at least this lets us identify and prevent some common user errors.

```
solidity import { SafeERC20 } from "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
```

The ERC-20 standard supports two ways for a contract to report failure:

1. Revert
2. Return `false`

Handling both cases would make our code more complicated, so instead we useOpenZeppelin's `SafeERC20`, which makes sure all failures result in a revert.

```
solidity /** * @title L1StandardBridge * @dev The L1 ETH and ERC20 Bridge is a contract which stores deposited L1
funds and standard * tokens that are in use on L2. It synchronizes a corresponding L2 Bridge, informing it of
deposits * and listening to it for newly finalized withdrawals. * */ contract L1StandardBridge is IL1StandardBridge,
CrossDomainEnabled { using SafeERC20 for IERC20;
```

This line is how we specify to use the`SafeERC20` wrapper every time we use the`IERC20` interface.

```solidity
```

```
/*******************************
 * External Contract References *
 *******************************/

address public l2TokenBridge;

```

The address of L2StandardBridge.

```solidity

```
// Maps L1 token to L2 token to balance of the L1 token deposited
mapping(address => mapping(address => uint256)) public deposits;

```

A double mapping like this is the way you define a two-dimensional sparse array. Values in this data structure are identified as `deposit[L1 token addr][L2 token addr]`. The default value is zero. Only cells that are set to a different value are written to storage.

```solidity

```
/***************
 * Constructor *
 ***************/

// This contract lives behind a proxy, so the constructor parameters will go unused.
constructor() CrossDomainEnabled(address(0)) {}

```

To want to be able to upgrade this contract without having to copy all the variables in the storage. To do that we use a `proxy`, a contract that uses `delegatecall` to transfer calls to a separate contact whose address is stored by the proxy contract (when you upgrade you tell the proxy to change that address). When you use `delegatecall` the storage remains the storage of the *calling* contract, so the values of all the contract state variables are unaffected.

One effect of this pattern is that the storage of the contract that is the *called* of `delegatecall` is not used and therefore the constructor values passed to it do not matter. This is the reason we can provide a nonsensical value to the `CrossDomainEnabled` constructor. It is also the reason the initialization below is separate from the constructor.

```solidity /*** * ***Initialization*** * ***/

```
/**
 * @param _l1messenger L1 Messenger address being used for cross-chain communications.
 * @param _l2TokenBridge L2 standard bridge address.
 */
// slither-disable-next-line external-function

```

This Slither test identifies functions that are not called from the contract code and could therefore be declared `external` instead of `public`. The gas cost of `external` functions can be lower, because they can be provided with parameters in the calldata. Functions declared `public` have to be accessible from within the contract. Contracts cannot modify their own calldata, so the parameters have to be in memory. When such a function is called externally, it is necessary to copy the calldata to memory, which costs gas. In this case the function is only called once, so the inefficiency does not matter to us.

`solidity function initialize(address _l1messenger, address _l2TokenBridge) public { require(messenger == address(0), "Contract has already been initialized.");`

The `initialize` function should only be called once. If the address of either the L1 cross domain messenger or the L2 token bridge changes, we create a new proxy and a new bridge that calls it. This is unlikely to happen except when the entire system is upgraded, a very rare occurrence.

Note that this function does not have any mechanism that restricts *who* can call it. This means that in theory an attacker could wait until we deploy the proxy and the first version of the bridge and then front-run to get to the `initialize` function before the legitimate user does. But there are two methods to prevent this:

1. If the contracts are deployed not directly by an EOA but in a transaction that has another contract create them the entire
```

process can be atomic, and finish before any other transaction is executed.
2. If the legitimate call to `initialize` fails it is always possible to ignore the newly created proxy and bridge and create new ones.

```
solidity messenger = _l1messenger; l2TokenBridge = _l2TokenBridge; }
```

These are the two parameters that the bridge needs to know.

```solidity

/**************
 * Depositing *
 **************/

/** @dev Modifier requiring sender to be EOA.  This check could be bypassed by a malicious
 *  contract via initcode, but it takes care of the user error we want to avoid.
 */
modifier onlyEOA() {
    // Used to stop deposits from contracts (avoid accidentally lost tokens)
    require(!Address.isContract(msg.sender), "Account not EOA");
    _;
}

```

This is the reason we needed OpenZeppelin's`Address` utilities.

```
solidity /** * @dev This function can be called with no data * to deposit an amount of ETH to the caller's balance
on L2. * Since the receive function doesn't take data, a conservative * default amount is forwarded to L2. */
receive() external payable onlyEOA { _initiateETHDeposit(msg.sender, msg.sender, 200_000, bytes("")); }
```

This function exists for testing purposes. Notice that it doesn't appear in the interface definitions - it isn't for normal use.

```solidity /* * *@inheritdoc IL1StandardBridge* / function depositETH(uint32 _l2Gas, bytes calldata _data) external payable onlyEOA { _initiateETHDeposit(msg.sender, msg.sender, _l2Gas, _data); }

```
/**
 * @inheritdoc IL1StandardBridge
 */
function depositETHTo(
    address _to,
    uint32 _l2Gas,
    bytes calldata _data
) external payable {
    _initiateETHDeposit(msg.sender, _to, _l2Gas, _data);
}

```

These two functions are wrappers around`_initiateETHDeposit`, the function that handles the actual ETH deposit.

```
solidity /** * @dev Performs the logic for deposits by storing the ETH and informing the L2 ETH Gateway of * the
deposit. * @param _from Account to pull the deposit from on L1. * @param _to Account to give the deposit to on L2. *
@param _l2Gas Gas limit required to complete the deposit on L2. * @param _data Optional data to forward to L2. This
data is provided * solely as a convenience for external contracts. Aside from enforcing a maximum * length, these
contracts provide no guarantees about its content. */ function _initiateETHDeposit(address _from, address _to,
uint32 _l2Gas, bytes memory _data) internal { // Construct calldata for finalizeDeposit call bytes memory message
= abi.encodeWithSelector(
```

The way that cross domain messages work is that the destination contract is called with the message as its calldata. Solidity contracts always interpret their calldata is accordance with the ABI specifications. The Solidity function `abi.encodeWithSelector` creates that calldata.

```
solidity IL2ERC20Bridge.finalizeDeposit.selector, address(0), Lib_PredeployAddresses.OVM_ETH, _from, _to,
msg.value, _data );
```

The message here is to call the `finalizeDeposit` function with these parameters:

| Parameter | Value | Meaning | | --------- | ---------- | ---------- | | _l1Token | address(0) | Special value to stand for ETH (which isn't an ERC-20 token) on L1 | | _l2Token | Lib_PredeployAddresses.OVM_ETH | The L2 contract that manages ETH on Optimism, `0xDeadDeAddeAddEAddeadDEaDDEAdDeaDDeAD0000` (this contract is for internal Optimism use only) | | _from | _from | The address on L1 that sends the ETH | | _to | _to | The address on L2 that receives the ETH | | amount | msg.value | Amount of wei sent (which
```

has already been sent to the bridge) | | _data | _data | Additional date to attach to the deposit |

```
solidity // Send calldata into L2 // slither-disable-next-line reentrancy-events
sendCrossDomainMessage(l2TokenBridge, _l2Gas, message);
```

Send the message through the cross domain messenger.

```
solidity // slither-disable-next-line reentrancy-events emit ETHDepositInitiated(_from, _to, msg.value, _data); }
```

Emit an event to inform any decentralized application that listens of this transfer.

```solidity /* * *@inheritdoc IL1ERC20Bridge* / function depositERC20( . . . ) external virtual onlyEOA { _initiateERC20Deposit(_l1Token, _l2Token, msg.sender, msg.sender, _amount, _l2Gas, _data); }

```

```
/**
 * @inheritdoc IL1ERC20Bridge
 */
function depositERC20To(
    .
    .
    .
) external virtual {
    _initiateERC20Deposit(_l1Token, _l2Token, msg.sender, _to, _amount, _l2Gas, _data);
}

```

These two functions are wrappers around`_initiateERC20Deposit`, the function that handles the actual ERC-20 deposit.

```
solidity /** * @dev Performs the logic for deposits by informing the L2 Deposited Token * contract of the deposit and calling a handler to lock the L1 funds. (e.g. transferFrom) * * @param _l1Token Address of the L1 ERC20 we are depositing * @param _l2Token Address of the L1 respective L2 ERC20 * @param _from Account to pull the deposit from on L1 * @param _to Account to give the deposit to on L2 * @param _amount Amount of the ERC20 to deposit. * @param _l2Gas Gas limit required to complete the deposit on L2. * @param _data Optional data to forward to L2. This data is provided * solely as a convenience for external contracts. Aside from enforcing a maximum * length, these contracts provide no guarantees about its content. */ function _initiateERC20Deposit( address _l1Token, address _l2Token, address _from, address _to, uint256 _amount, uint32 _l2Gas, bytes calldata _data ) internal {
```

This function is similar to`_initiateETHDeposit` above, with a few important differences. The first difference is that this function receives the token addresses and the amount to transfer as parameters. In the case of ETH the call to the bridge already includes the transfer of asset to the bridge account (`msg.value`).

```
solidity // When a deposit is initiated on L1, the L1 Bridge transfers the funds to itself for future // withdrawals. safeTransferFrom also checks if the contract has code, so this will fail if // _from is an EOA or address(0). // slither-disable-next-line reentrancy-events, reentrancy-benign
IERC20(_l1Token).safeTransferFrom(_from, address(this), _amount);
```

ERC-20 token transfers follow a different process from ETH:

1. The user (`_from`) gives an allowance to the bridge to transfer the appropriate tokens.
2. The user calls the bridge with the address of the token contract, the amount, etc.
3. The bridge transfers the tokens (to itself) as part of the deposit process.

The first step may happen in a separate transaction from the last two. However, front-running is not a problem because the two functions that call `_initiateERC20Deposit` (`depositERC20` and `depositERC20To`) only call this function with`msg.sender` as the `_from` parameter.

```solidity // Construct calldata for _l2Token.finalizeDeposit(_to, _amount) bytes memory message = abi.encodeWithSelector( IL2ERC20Bridge.finalizeDeposit.selector, _l1Token, _l2Token, _from, _to, _amount, _data );

```

```
    // Send calldata into L2
    // slither-disable-next-line reentrancy-events, reentrancy-benign
    sendCrossDomainMessage(l2TokenBridge, _l2Gas, message);

    // slither-disable-next-line reentrancy-benign
    deposits[_l1Token][_l2Token] = deposits[_l1Token][_l2Token] + _amount;

```

Add the deposited amount of tokens to the`deposits` data structure. There could be multiple addresses on L2 that correspond to the same L1 ERC-20 token, so it is not sufficient to use the bridge's balance of the L1 ERC-20 token to keep track of deposits.

```solidity
    // slither-disable-next-line reentrancy-events
    emit ERC20DepositInitiated(_l1Token, _l2Token, _from, _to, _amount, _data);
}

/**********************
 * Cross-chain Functions *
 **********************/

/**
 * @inheritdoc IL1StandardBridge
 */
function finalizeETHWithdrawal(
    address _from,
    address _to,
    uint256 _amount,
    bytes calldata _data

```

The L2 bridge sends a message to the L2 cross domain messenger which causes the L1 cross domain messenger to call this function (once the transaction that finalizes the message is submitted on L1, of course).

```
solidity ) external onlyFromCrossDomainAccount(l2TokenBridge) {
```

Make sure that this is a *legitimate* message, coming from the cross domain messenger and originating with the L2 token bridge. This function is used to withdraw ETH from the bridge, so we have to make sure it is only called by the authorized caller.

```
solidity // slither-disable-next-line reentrancy-events (bool success, ) = _to.call{ value: _amount }(new
bytes(0));
```

The way to transfer ETH is to call the recipient with the amount of wei in the `msg.value`.

```solidity require(success, "TransferHelper::safeTransferETH: ETH transfer failed");

    // slither-disable-next-line reentrancy-events
    emit ETHWithdrawalFinalized(_from, _to, _amount, _data);

```

Emit an event about the withdrawal.

```solidity }

/**
 * @inheritdoc IL1ERC20Bridge
 */
function finalizeERC20Withdrawal(
    address _l1Token,
    address _l2Token,
    address _from,
    address _to,
    uint256 _amount,
    bytes calldata _data
) external onlyFromCrossDomainAccount(l2TokenBridge) {

```

This function is similar to `finalizeETHWithdrawal` above, with the necessary changes for ERC-20 tokens.

```
solidity deposits[_l1Token][_l2Token] = deposits[_l1Token][_l2Token] - _amount;
```

Update the `deposits` data structure.

```solidity
    // When a withdrawal is finalized on L1, the L1 Bridge transfers the funds to the withdrawer
    // slither-disable-next-line reentrancy-events
    IERC20(_l1Token).safeTransfer(_to, _amount);

    // slither-disable-next-line reentrancy-events
    emit ERC20WithdrawalFinalized(_l1Token, _l2Token, _from, _to, _amount, _data);
}
```

```
/*****************************
 * Temporary - Migrating ETH *
 *****************************/

/**
 * @dev Adds ETH balance to the account. This is meant to allow for ETH
 * to be migrated from an old gateway to a new gateway.
 * NOTE: This is left for one upgrade only so we are able to receive the migrated ETH from the
 * old contract
 */
function donateETH() external payable {}

}
```

There was an earlier implementation of the bridge. When we moved from the implementation to this one, we had to move all the assets. ERC-20 tokens can just be moved. However, to transfer ETH to a contract you need that contract's approval, which is what `donateETH` provides us.

ERC-20 Tokens on L2 {#erc-20-tokens-on-l2}

For an ERC-20 token to fit into the standard bridge, it needs to allow the standard bridge, and *only* the standard bridge, to mint token. This is necessary because the bridges need to ensure that the number of tokens circulating on Optimism is equal to the number of tokens locked inside the L1 bridge contract. If there are too many tokens on L2 some users would be unable to bridge their assets back to L1. Instead of a trusted bridge, we would essentially recreate fractional reserve banking. If there are too many tokens on L1, some of those tokens would stay locked inside the bridge contract forever because there is no way to release them without burning L2 tokens.

IL2StandardERC20 {#il2standarderc20}

Every ERC-20 token on L2 that uses the standard bridge needs to provide this interface, which has the functions and events that the standard bridge needs.

```solidity // SPDX-License-Identifier: MIT pragma solidity ^0.8.9;

import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol"; ```

The standard ERC-20 interface does not include the `mint` and `burn` functions. Those methods are not required by the ERC-20 standard, which leaves unspecified the mechanisms to create and destroy tokens.

```
solidity import { IERC165 } from "@openzeppelin/contracts/utils/introspection/IERC165.sol";
```

The ERC-165 interface is used to specify what functions a contract provides. You can read the standard here.

```
solidity interface IL2StandardERC20 is IERC20, IERC165 { function l1Token() external returns (address);
```

This function provides the address of the L1 token which is bridged to this contract. Note that we do not have a similar function in the opposite direction. We need to be able to bridge any L1 token, regardless of whether L2 support was planned when it was implemented or not.

```solidity

```
function mint(address _to, uint256 _amount) external;

function burn(address _from, uint256 _amount) external;

event Mint(address indexed _account, uint256 _amount);
event Burn(address indexed _account, uint256 _amount);
```

}
```

Functions and events to mint (create) and burn (destroy) tokens. The bridge should be the only entity that can run these functions to ensure the number of tokens is correct (equal to the number of tokens locked on L1).

## L2StandardERC20 {#L2StandardERC20}

This is our implementation of the `IL2StandardERC20` interface. Unless you need some kind of custom logic, you should use this one.
```

```solidity // SPDX-License-Identifier: MIT pragma solidity ^0.8.9;

import { ERC20 } from "@openzeppelin/contracts/token/ERC20/ERC20.sol"; ```

The OpenZeppelin ERC-20 contract. Optimism does not believe in reinventing the wheel, especially when the wheel is well audited and needs to be trustworthy enough to hold assets.

```solidity import "./IL2StandardERC20.sol";

contract L2StandardERC20 is IL2StandardERC20, ERC20 { address public l1Token; address public l2Bridge; ```

These are the two additional configuration parameters that we require and ERC-20 normally does not.

```solidity

```
/**
 * @param _l2Bridge Address of the L2 standard bridge.
 * @param _l1Token Address of the corresponding L1 token.
 * @param _name ERC20 name.
 * @param _symbol ERC20 symbol.
 */
constructor(
    address _l2Bridge,
    address _l1Token,
    string memory _name,
    string memory _symbol
) ERC20(_name, _symbol) {
    l1Token = _l1Token;
    l2Bridge = _l2Bridge;
}
```

```

First call the constructor for the contract we inherit from (`ERC20(_name, _symbol)`) and then set our own variables.

```solidity

```
modifier onlyL2Bridge() {
    require(msg.sender == l2Bridge, "Only L2 Bridge can mint and burn");
    _;
}

// slither-disable-next-line external-function
function supportsInterface(bytes4 _interfaceId) public pure returns (bool) {
    bytes4 firstSupportedInterface = bytes4(keccak256("supportsInterface(bytes4)")); // ERC165
    bytes4 secondSupportedInterface = IL2StandardERC20.l1Token.selector ^
        IL2StandardERC20.mint.selector ^
        IL2StandardERC20.burn.selector;
    return _interfaceId == firstSupportedInterface || _interfaceId == secondSupportedInterface;
}
```

```

This is the way ERC-165 works. Every interface is a number of supported functions, and is identified as theexclusive or of the ABI function selectors of those functions.

The L2 bridge uses ERC-165 as a sanity check to make sure that the ERC-20 contract to which it sends assets is an `IL2StandardERC20`.

Note: There is nothing to prevent rogue contract from providing false answers to`supportsInterface`, so this is a sanity check mechanism, *not* a security mechanism.

```solidity // slither-disable-next-line external-function function mint(address _to, uint256 _amount) public virtual onlyL2Bridge { _mint(_to, _amount);

```
    emit Mint(_to, _amount);
}

// slither-disable-next-line external-function
function burn(address _from, uint256 _amount) public virtual onlyL2Bridge {
    _burn(_from, _amount);

    emit Burn(_from, _amount);
```
```

```
}
```

} ```

Only the L2 bridge is allowed to mint and burn assets.

`_mint` and `_burn` are actually defined in the[OpenZeppelin ERC-20 contract](). That contract just doesn't expose them externally, because the conditions to mint and burn tokens are as varied as the number of ways to use ERC-20.

L2 Bridge Code {#l2-bridge-code}

This is code that runs the bridge on Optimism.[The source for this contract is here]().

```solidity // SPDX-License-Identifier: MIT pragma solidity ^0.8.9;

/ *Interface Imports* / import { IL1StandardBridge } from "../../L1/messaging/IL1StandardBridge.sol"; import { IL1ERC20Bridge } from "../../L1/messaging/IL1ERC20Bridge.sol"; import { IL2ERC20Bridge } from "./IL2ERC20Bridge.sol"; ```

The [IL2ERC20Bridge]() interface is very similar to the[L1 equivalent]() we saw above. There are two significant differences:

1. On L1 you initiate deposits and finalize withdrawals. Here you initiate withdrawals and finalize deposits.
2. On L1 it is necessary to distinguish between ETH and ERC-20 tokens. On L2 we can use the same functions for both because internally ETH balances on Optimism are handled as an ERC-20 token with the address [0xDeadDeAddeAddEAddeadDEaDDEAdDeaDDeAD0000]().

```solidity / *Library Imports* / import { ERC165Checker } from "@openzeppelin/contracts/utils/introspection/ERC165Checker.sol"; import { CrossDomainEnabled } from "../../libraries/bridge/CrossDomainEnabled.sol"; import { Lib_PredeployAddresses } from "../../libraries/constants/Lib_PredeployAddresses.sol";

/ *Contract Imports* / import { IL2StandardERC20 } from "../../standards/IL2StandardERC20.sol";

/ *** @title L2StandardBridge * @dev The L2 Standard bridge is a contract which works together with the L1 Standard bridge to * enable ETH and ERC20 transitions between L1 and L2. * This contract acts as a minter for new tokens when it hears about deposits into the L1 Standard * bridge. * This contract also acts as a burner of the tokens intended for withdrawal, informing the L1 * bridge to release L1 funds.** */ ***contract L2StandardBridge is IL2ERC20Bridge, CrossDomainEnabled {*** /**** * External Contract References * *******/

```
address public l1TokenBridge;
```

```

Keep track of the address of the L1 bridge. Note that in contrast to the L1 equivalent, here we*need* this variable. The address of the L1 bridge is not known in advance.

```solidity
```
/***************
 * Constructor *
 ***************/

/**
 * @param _l2CrossDomainMessenger Cross-domain messenger used by this contract.
 * @param _l1TokenBridge Address of the L1 bridge deployed to the main chain.
 */
constructor(address _l2CrossDomainMessenger, address _l1TokenBridge)
    CrossDomainEnabled(_l2CrossDomainMessenger)
{
    l1TokenBridge = _l1TokenBridge;
}

/***************
 * Withdrawing *
 ***************/

/**
 * @inheritdoc IL2ERC20Bridge
 */
function withdraw(
    address _l2Token,
```

```
    uint256 _amount,
    uint32 _l1Gas,
    bytes calldata _data
) external virtual {
    _initiateWithdrawal(_l2Token, msg.sender, msg.sender, _amount, _l1Gas, _data);
}

/**
 * @inheritdoc IL2ERC20Bridge
 */
function withdrawTo(
    address _l2Token,
    address _to,
    uint256 _amount,
    uint32 _l1Gas,
    bytes calldata _data
) external virtual {
    _initiateWithdrawal(_l2Token, msg.sender, _to, _amount, _l1Gas, _data);
}

```

These two functions initiate withdrawals. Note that there is no needs to specify the L1 token address. L2 tokens are expected to tell us the L1 equivalent's address.

```solidity
/**
 * @dev Performs the logic for withdrawals by burning the token and informing
 *      the L1 token Gateway of the withdrawal.
 * @param _l2Token Address of L2 token where withdrawal is initiated.
 * @param _from Account to pull the withdrawal from on L2.
 * @param _to Account to give the withdrawal to on L1.
 * @param _amount Amount of the token to withdraw.
 * @param _l1Gas Unused, but included for potential forward compatibility considerations.
 * @param _data Optional data to forward to L1. This data is provided
 *        solely as a convenience for external contracts. Aside from enforcing a maximum
 *        length, these contracts provide no guarantees about its content.
 */
function _initiateWithdrawal(
    address _l2Token,
    address _from,
    address _to,
    uint256 _amount,
    uint32 _l1Gas,
    bytes calldata _data
) internal {
    // When a withdrawal is initiated, we burn the withdrawer's funds to prevent subsequent L2
    // usage
    // slither-disable-next-line reentrancy-events
    IL2StandardERC20(_l2Token).burn(msg.sender, _amount);

```

Notice that we are *not* relying on the `_from` parameter but on `msg.sender` which is a lot harder to fake (impossible, as far as I know).

```solidity
    // Construct calldata for l1TokenBridge.finalizeERC20Withdrawal(_to, _amount)
    // slither-disable-next-line reentrancy-events
    address l1Token = IL2StandardERC20(_l2Token).l1Token();
    bytes memory message;

    if (_l2Token == Lib_PredeployAddresses.OVM_ETH) {

```

On L1 it is necessary to distinguish between ETH and ERC-20.

```solidity message = abi.encodeWithSelector( IL1StandardBridge.finalizeETHWithdrawal.selector, _from, _to, _amount, _data ); } else { message = abi.encodeWithSelector( IL1ERC20Bridge.finalizeERC20Withdrawal.selector, l1Token, _l2Token, _from, _to, _amount, _data ); }

    // Send message up to L1 bridge
    // slither-disable-next-line reentrancy-events
    sendCrossDomainMessage(l1TokenBridge, _l1Gas, message);
```

```
    // slither-disable-next-line reentrancy-events
    emit WithdrawalInitiated(l1Token, _l2Token, msg.sender, _to, _amount, _data);
}

/**********************************
 * Cross-chain Function: Depositing *
 **********************************/

/**
 * @inheritdoc IL2ERC20Bridge
 */
function finalizeDeposit(
    address _l1Token,
    address _l2Token,
    address _from,
    address _to,
    uint256 _amount,
    bytes calldata _data

```

This function is called by `L1StandardBridge`.

```
solidity ) external virtual onlyFromCrossDomainAccount(l1TokenBridge) {
```

Make sure the source of the message is legitimate. This is important because this function calls `_mint` and could be used to give tokens that are not covered by tokens the bridge owns on L1.

```
solidity // Check the target token is compliant and // verify the deposited token on L1 matches the L2 deposited
token representation here if ( // slither-disable-next-line reentrancy-events
ERC165Checker.supportsInterface(_l2Token, 0x1d1d8b63) && _l1Token == IL2StandardERC20(_l2Token).l1Token()
```

Sanity checks:

1. The correct interface is supported
2. The L2 ERC-20 contract's L1 address matches the L1 source of the tokens

```
solidity ) { // When a deposit is finalized, we credit the account on L2 with the same amount of // tokens. //
slither-disable-next-line reentrancy-events IL2StandardERC20(_l2Token).mint(_to, _amount); // slither-disable-next-
line reentrancy-events emit DepositFinalized(_l1Token, _l2Token, _from, _to, _amount, _data);
```

If the sanity checks pass, finalize the deposit:

1. Mint the tokens
2. Emit the appropriate event

```
solidity } else { // Either the L2 token which is being deposited-into disagrees about the correct address // of its
L1 token, or does not support the correct interface. // This should only happen if there is a malicious L2 token, or
if a user somehow // specified the wrong L2 token address to deposit into. // In either case, we stop the process
here and construct a withdrawal // message so that users can get their funds out in some cases. // There is no way
to prevent malicious token contracts altogether, but this does limit // user error and mitigate some forms of
malicious contract behavior.
```

If a user made a detectable error by using the wrong L2 token address, we want to cancel the deposit and return the tokens on L1. The only way we can do this from L2 is to send a message that will have to wait the fault challenge period, but that is much better for the user than losing the tokens permanently.

```solidity bytes memory message = abi.encodeWithSelector( IL1ERC20Bridge.finalizeERC20Withdrawal.selector, _l1Token, _l2Token, _to, // switched the _to and _from here to bounce back the deposit to the sender _from, _amount, _data );

        // Send message up to L1 bridge
        // slither-disable-next-line reentrancy-events
        sendCrossDomainMessage(l1TokenBridge, 0, message);
        // slither-disable-next-line reentrancy-events
        emit DepositFailed(_l1Token, _l2Token, _from, _to, _amount, _data);
    }
}

} ```

## Conclusion {#conclusion}
```

The standard bridge is the most flexible mechanism for asset transfers. However, because it is so generic it is not always the easiest mechanism to use. Especially for withdrawals, most users prefer to use third party bridges that do not wait the challenge period and do not require a Merkle proof to finalize the withdrawal.

These bridges typically work by having assets on L1, which they provide immediately for a small fee (often less than the cost of gas for a standard bridge withdrawal). When the bridge (or the people running it) anticipates being short on L1 assets it transfers sufficient assets from L2. As these are very big withdrawals, the withdrawal cost is amortized over a large amount and is a much smaller percentage.

Hopefully this article helped you understand more about how layer 2 works, and how to write Solidity code that is clear and secure.