Communicating with L1

Is this your first time hearing the wordPortal ? Check out theconcepts section .

Follow thetoken bridge tutorial for hands-on experience writing and deploying a Portal contract.

Passing data to the rollup

Whether it is tokens or other information being passed to the rollup, the portal should use theInbox to do it.

TheInbox can be seen as a mailbox to the rollup, portals put messages into the box, and the sequencers then decide which of these message they want to include in their blocks (each message has a fee attached to it, so there is a fee market here).

When sending messages, we need to specify quite a bit of information beyond just the content that we are sharing. Namely we need to specify:

Name Type Description Recipient L2Actor The message recipient. ThisMUST match the rollup version and an Aztec contract that isattached to the contract making this call. If the recipient is not attached to the caller, the message cannot be consumed by it. Deadline uint256 The deadline for the message to be consumed. If the message has not been removed from theInbox and included in a rollup block by this point, it can becanceled by the portal (the portal must implement logic to cancel). Content field (~254 bits) The content of the message. This is the data that will be passed to the recipient. The content is limited to be a single field. If the content is small enough it can just be passed along, otherwise it should be hashed and the hash passed along (you can use ourHash utilities withsha256ToField functions) Secret Hash field (~254 bits) A hash of a secret that is used when consuming the message on L2. Keep this preimage a secret to make the consumption private. To consume the message the caller must know the pre-image (the value that was hashed) - so make sure your app keeps track of the pre-images! Use thecomputeMessageSecretHash to compute it from a secret. Fee uint64 The fee to the sequencer for including the message. This is the amount of ETH that the sequencer will receive for including the message. Note that it is not a fulluint256 but onlyuint64 With all that information at hand, we can call thesendL2Message function on the Inbox. The function will return afield (insidebytes32) that is the hash of the message. This hash can be used as an identifier to spot when your message has been included in a rollup block.

send_l1_to_l2_message /* *@notice Inserts an entry into the Inbox * @dev Will emitMessageAdded with data for easy access by the sequencer * @dev msg.value - The fee provided to sequencer for including the entry * @param _recipient - The recipient of the entry * @param _deadline - The deadline to consume a message. Only after it, can a message be cancelled. * @param _content - The content of the entry (application specific) * @param _secretHash - The secret hash of the entry (make it possible to hide when a specific entry is consumed on L2) * @return The key of the entry in the set* / function

sendL2Message (DataStructures . L2Actor memory _recipient , uint32 _deadline , bytes32 _content , bytes32 _secretHash)

external

payable

returns

(bytes32) ;Source code: l1-contracts/src/core/interfaces/messagebridge/IInbox.sol#L27-L44 As time passes, a sequencer will see your tx, the juicy fee provided and include it in a rollup block. Upon inclusion, it is removed from L1, and made available to be consumed on L2.

To consume the message, we can use theconsume_l1_to_l2_message function within thecontext struct.

- Themsg_key
- is the hash of the message returned by thesendL2Message
- call and is used to help the RPC find the correct message.
- Thecontent
- is the content of the message, limited to one Field element. For content larger than one Field, we suggest using thesha256
- hash function truncated to a single Field element.sha256
- is suggested as it is cheap on L1 while still being manageable on L2.
- Thesecret
- is the pre-image hashed using Pedersen to compute thesecretHash
- .
- If thecontent
- orsecret
- does not match the entry atmsg_key
- the message will not be consumed, and the transaction will revert.

info Note that while thesecret and thecontent are both hashed, they are actually hashed with different hash functions! context_consume_l1_to_l2_message pub

fn

consume_l1_to_l2_message (& mut

self , content :

Field , secret :

Field , sender :

EthAddress)

{ Source code: noir-projects/aztec-nr/aztec/src/context/private_context.nr#L229-L232 Computing thecontent must be done manually in its current form, as we are still adding a number of bytes utilities. A good example exists within theToken bridge example .

claim_public // Consumes a L1->L2 message and calls the token contract to mint the appropriate amount publicly

[aztec(public)]

fn

claim_public (to :

AztecAddress , amount :

Field , canceller :

EthAddress , secret :

Field)

{ let content_hash =

get_mint_public_content_hash (to , amount , canceller) ;

// Consume message and emit nullifier context . consume_l1_to_l2_message (content_hash , secret , context . this_portal_address ( )) ;

// Mint tokens Token :: at (storage . token . read ( )) . mint_public (& mut context , to , amount) } Source code: noir-projects/noir-contracts/contracts/token_bridge_contract/src/main.nr#L34-L46 info Thecontent_hash is a sha256 truncated to a field element (~ 254 bits). In Aztec-nr, you can use oursha256_to_field() to do a sha256 hash which fits in one field element mint_public_content_hash_nr use

dep :: aztec :: prelude :: { AztecAddress ,

EthAddress } ; use

dep :: aztec :: protocol_types :: hash :: sha256_to_field ;

// Computes a content hash of a deposit/mint_public message. // Refer TokenPortal.sol for reference on L1. pub

fn

get_mint_public_content_hash (owner :

AztecAddress , amount :

Field , canceller :

EthAddress)

->

Field

{ let

mut hash_bytes :

[u8 ;

100]

=

[0 ;

100] ; let recipient_bytes = owner . to_field ( ) . to_be_bytes (32) ; let amount_bytes = amount . to_be_bytes (32) ; let canceller_bytes = canceller . to_field ( ) . to_be_bytes (32) ;

for i in

0 .. 32

{ hash_bytes [i +

4]

= recipient_bytes [i] ; hash_bytes [i +

36]

= amount_bytes [i] ; hash_bytes [i +

68]

= canceller_bytes [i] ; }

// Function selector: 0xefc2aae6 keccak256('mint_public(bytes32,uint256,address)') hash_bytes [0]

=

0xef ; hash_bytes [1]

=

0xc2 ; hash_bytes [2]

=

0xaa ; hash_bytes [3]

=

0xe6 ;

let content_hash =

sha256_to_field (hash_bytes) ; content_hash }Source code: noir-projects/noir-contracts/contracts/token_portal_content_hash_lib/src/lib.nr#L1-L28 In Solidity, you can use ourHash.sha256ToField() method:

content_hash_sol_import import

{ Hash }

from

"../../src/core/libraries/Hash.sol" ; Source code: l1-contracts/test/portals/TokenPortal.sol#L10-L12 deposit_public /* * *@notice Deposit funds into the portal and adds an L2 message which can only be consumed publicly on Aztec * @param _to - The aztec address of the recipient * @param _amount - The amount to deposit * @param _canceller - The address that can cancel the L1 to L2 message * @param _deadline - The timestamp after which the entry can be cancelled * @param _secretHash - The hash of the secret consumable message. The hash should be 254 bits (so it can fit in a Field element) * @return The key of the entry in the Inbox* / function

depositToAztecPublic (bytes32 _to , uint256 _amount , address _canceller , uint32 _deadline , bytes32 _secretHash)

external

payable

returns

(bytes32)

{ // Preamble IInbox inbox = registry . getInbox ( ) ; DataStructures . L2Actor memory actor = DataStructures . L2Actor (l2TokenAddress ,

1) ;

// Hash the message content to be reconstructed in the receiving contract bytes32 contentHash = Hash . sha256ToField (abi . encodeWithSignature ("mint_public(bytes32,uint256,address)" , _to , _amount , _canceller)) ;

// Hold the tokens in the portal underlying . safeTransferFrom (msg . sender ,

address (this) , _amount) ;

// Send message to rollup return inbox . sendL2Message { value : msg . value } (actor , _deadline , contentHash , _secretHash) ; } Source code: l1-contracts/test/portals/TokenPortal.sol#L29-L61 Thesecret_hash uses the pederson hash which fits in a field element. You can use the utility methodcomputeMessageSecretHash() in@aztec/aztec.js npm package to generate a secret and its corresponding hash.

After the transaction has been mined, the message is consumed, a nullifier is emitted and the tokens have been minted on Aztec and are ready for claiming.

Since the message consumption is emitting a nullifier the same message cannot be consumed again. The index in the message tree is used as part of the nullifier computation, ensuring that the same content and secret being inserted will be distinct messages that can each be consumed. Without the index in the nullifier, it would be possible to perform a kind of attack known asFaerie Gold attacks where two seemingly good messages are inserted, but only one of them can be consumed later.

Passing data to L1

To pass data to L1, we use theOutbox . TheOutbox is the mailbox for L2 to L1 messages. This is the location on L1 where all the messages from L2 will live, and where they can be consumed from.

Similarly to messages going to L2 from L1, a message can only be consumed by the recipient, however note that it is up to the portal contract to ensure that the sender is as expected!

Recall that we mentioned the Aztec contract specifies what portal it is attached to at deployment. This value is stored in the rollup's contract tree, hence these links are not directly readable on L1. Also, it is possible to attach multiple aztec contracts to the same portal.

The portal must ensure that the sender is as expected. One way to do this is to compute the addresses before deployment and store them as constants in the contract. However, a more flexible solution is to have aninitialize function in the portal contract which can be used to set the address of the Aztec contract. In this model, the portal contract can check that the sender matches the value it has in storage.

To send a message to L1 from your Aztec contract, you must use themessage_portal function on thecontext . When messaging to L1, only thecontent is required (as aField).

context_message_portal pub

fn

message_portal (& mut

self , recipient :

EthAddress , content :

Field)

{ Source code: noir-projects/aztec-nr/aztec/src/context/private_context.nr#L222-L224 When sending a message from L2 to L1 we don't need to pass recipient, deadline, secret nor fees. Recipient is populated with the attached portal and the remaining values are not needed as the message is inserted into the outbox at the same time as it was included in a block (for the inbox it could be inserted and then only included in rollup block later).

danger Access control on the L1 portal contract is essential to prevent consumption of messages sent from the wrong L2 contract. As earlier, we can use a token bridge as an example. In this case, we are burning tokens on L2 and sending a message to the portal to free them on L1.

exit_to_l1_private // Burns the appropriate amount of tokens and creates a L2 to L1 withdraw message privately // Requires

msg.sender (caller of the method) to give approval to the bridge to burn tokens on their behalf using witness signatures

[aztec(private)]

fn

exit_to_l1_private (token :

AztecAddress , recipient :

EthAddress ,

// ethereum address to withdraw to amount :

Field , callerOnL1 :

EthAddress ,

// ethereum address that can call this function on the L1 portal (0x0 if anyone can call) nonce :

Field

// nonce used in the approval message bymsg.sender to let bridge burn their tokens on L2)

{ // Send an L2 to L1 message let content =

get_withdraw_content_hash (recipient , amount , callerOnL1) ; context . message_portal (context . this_portal_address ( ) , content) ;

// Assert that user provided token address is same as seen in storage. context . call_public_function (context . this_address ( ) , FunctionSelector :: from_signature ("_assert_token_is_same((Field))") , [token . to_field ( )]) ;

// Burn tokens Token :: at (token) . burn (& mut context , context . msg_sender ( ) , amount , nonce) ;Source code: noir-projects/noir-contracts/contracts/token_bridge_contract/src/main.nr#L96-L123 When the transaction is included in a rollup block the message will be inserted into theOutbox , where the recipient portal can consume it from. When consuming, themsg.sender must match therecipient meaning that only portal can actually consume the message.

l2_to_l1_msg /* *@notice Struct containing a message from L2 to L1 * @param sender - The sender of the message * @param recipient - The recipient of the message * @param content - The content of the message (application specific) padded to bytes32 or hashed if larger. * @dev Not to be confused with L2ToL1Message in Noir circuits* / struct

L2ToL1Msg

{ DataStructures . L2Actor sender ; DataStructures . L1Actor recipient ; bytes32 content ; }Source code: l1-contracts/src/core/libraries/DataStructures.sol#L71-L84 outbox_consume /* *@notice Consumes an entry from the Outbox * @dev Only meaningfully callable by portals, otherwise should never hit an entry * @dev Emits the MessageConsumed event when consuming messages * @param _message - The L2 to L1 message * @return entryKey - The key of the entry removed* / function

consume (DataStructures . L2ToL1Msg memory _message)

external

returns

(bytes32 entryKey) ;Source code: l1-contracts/src/core/interfaces/messagebridge/IOutbox.sol#L37-L46 As noted earlier, the portal contract should check that the sender is as expected. In the example below, we support only one sender contract (stored inl2TokenAddress) so we can just pass it as the sender, that way we will only be able to consume messages from that contract. If multiple senders are supported, you could use a havemapping(address => bool) allowed and check thatallowed[msg.sender] istrue .

token_portal_withdraw /* *@notice Withdraw funds from the portal * @dev Second part of withdraw, must be initiated from L2 first as it will consume a message from outbox * @param _recipient - The address to send the funds to * @param _amount - The amount to withdraw * @param _withCaller - Flag to use msg.sender as caller, otherwise address(0) * Must match the caller of the message (specified from L2) to consume it. * @return The key of the entry in the Outbox* / function

withdraw (address _recipient ,

uint256 _amount ,

bool _withCaller) external returns

(bytes32) { DataStructures . L2ToL1Msg memory message = DataStructures . L2ToL1Msg ({ sender : DataStructures . L2Actor (l2TokenAddress ,

1) , recipient : DataStructures . L1Actor (address (this) , block . chainid) , content : Hash . sha256ToField (abi . encodeWithSignature ("withdraw(address,uint256,address)" , _recipient , _amount , _withCaller ? msg . sender :

address (0))) }) ;

bytes32 entryKey = registry . getOutbox ( ) . consume (message) ;

underlying . transfer (_recipient , _amount) ;

return entryKey ; } Source code: l1-contracts/test/portals/TokenPortal.sol#L186-L219

Considerations

Structure of messages

The application developer should consider creating messages that follow a function call structure e.g., using a function signature and arguments. This will make it easier to prevent producing messages that could be misinterpreted by the recipient.

An example of a bad format would be usingamount, token_address, recipient_address as the message for a withdraw function andamount, token_address, on_behalf_of_address for a deposit function. Any deposit could then also be mapped to a withdraw or vice versa.

// Don't to this! bytes

memory message = abi . encode (_amount , _token , _to) ;

// Do this! bytes

memory message abi . encodeWithSignature ("withdraw(uint256,address,address)" , _amount , _token , _to) ;

Error Handling

Handling error when moving cross chain can quickly get tricky. Since the L1 and L2 calls are practically async and independent of each other, the L1 part of a deposit might execute just fine, with the L2 part failing. If this is not handled well, the funds may be lost forever! The contract builder should therefore consider ways their application can fail cross chain, and handle all cases explicitly.

First, entries in the outboxesSHOULD only be consumed if the execution is successful. For an L2 -> L1 call, the L1 execution can revert the transaction completely if anything fails. As the tx is atomic, the failure also reverts consumption.

If it is possible to enter a state where the second part of the execution fails forever, the application builder should consider including additional failure mechanisms (for token withdraws this could be depositing them again etc).

Generally it is good practice to keep cross-chain calls simple to avoid too many edge cases and state reversions.

info Error handling for cross chain messages is handled by the application contract and not the protocol. The protocol only delivers the messages, it does not ensure that they are executed successfully.

Cancellations

A special type of error is an underpriced transaction - it means that a message is inserted on L1, but the attached fee is too low to be included in a rollup block.

For the case of token bridges, this could lead to funds being locked in the bridge forever, as funds are locked but the message never arrives on L2 to mint the tokens. To address this, theInbox supports canceling messages after a deadline. However, this must be called by the portal itself, as it will need to "undo" the state changes is made (for example by sending the tokens back to the user).

As this requires logic on the portal itself, it is not something that the protocol can enforce. It must be supported by the application builder when building the portal.

The portal can call thecancelL2Message at theInbox whenblock.timestamp > deadline for the message.

pending_l2_cancel /* * @notice Cancel a pending L2 message * @dev Will revert if the deadline have not been crossed -

*message only cancellable past the deadline * so it cannot be yanked away while the sequencer is building a block including it * @dev Must be called by portal that inserted the entry * @param _message - The content of the entry (application specific) * @param _feeCollector - The address to receive the "fee" * @return entryKey - The key of the entry removed* / function

cancelL2Message (DataStructures . L1ToL2Msg memory _message ,

address _feeCollector) external returns

(bytes32 entryKey) ;Source code: l1-contracts/src/core/interfaces/messagebridge/IInbox.sol#L46-L59 Building on our token example from earlier, this can be called like:

token_portal_cancel /* * *@notice Cancel a public depositToAztec L1 to L2 message * @dev only callable by the _canceller of the message * @param _to - The aztec address of the recipient in the original message * @param _amount - The amount to deposit per the original message * @param _deadline - The timestamp after which the entry can be cancelled * @param _secretHash - The hash of the secret consumable message in the original message * @param _fee - The fee paid to the sequencer * @return The key of the entry in the Inbox* / function

cancelL1ToAztecMessagePublic (bytes32 _to , uint256 _amount , uint32 _deadline , bytes32 _secretHash , uint64 _fee)

external

returns

(bytes32)

{ IInbox inbox = registry . getInbox ( ) ; DataStructures . L1Actor memory l1Actor = DataStructures . L1Actor (address (this) , block . chainid) ; DataStructures . L2Actor memory l2Actor = DataStructures . L2Actor (l2TokenAddress ,

1) ; DataStructures . L1ToL2Msg memory message = DataStructures . L1ToL2Msg ({ sender : l1Actor , recipient : l2Actor , content : Hash . sha256ToField (abi . encodeWithSignature ("mint_public(bytes32,uint256,address)" , _to , _amount , msg . sender)) , secretHash : _secretHash , deadline : _deadline , fee : _fee }) ; bytes32 entryKey = inbox . cancelL2Message (message ,

address (this)) ; // release the funds to msg.sender (since the content hash (& entry key) is derived by hashing the caller, // we confirm that msg.sender is same as _canceller supplied when creating the message) underlying . transfer (msg . sender , _amount) ; return entryKey ; }

/* * *@notice Cancel a private depositToAztec L1 to L2 message * @dev only callable by the _canceller of the message * @param _secretHashForRedeemingMintedNotes - The hash of the secret to redeem minted notes privately on Aztec * @param _amount - The amount to deposit per the original message * @param _deadline - The timestamp after which the entry can be cancelled * @param _secretHashForL2MessageConsumption - The hash of the secret consumable L1 to L2 message * @param _fee - The fee paid to the sequencer * @return The key of the entry in the Inbox* / function

cancelL1ToAztecMessagePrivate (bytes32 _secretHashForRedeemingMintedNotes , uint256 _amount , uint32 _deadline , bytes32 _secretHashForL2MessageConsumption , uint64 _fee)

external

returns

(bytes32)

{ IInbox inbox = registry . getInbox ( ) ; DataStructures . L1Actor memory l1Actor = DataStructures . L1Actor (address (this) , block . chainid) ; DataStructures . L2Actor memory l2Actor = DataStructures . L2Actor (l2TokenAddress ,

1) ; DataStructures . L1ToL2Msg memory message = DataStructures . L1ToL2Msg ({ sender : l1Actor , recipient : l2Actor , content : Hash . sha256ToField (abi . encodeWithSignature ("mint_private(bytes32,uint256,address)" , _secretHashForRedeemingMintedNotes , _amount , msg . sender)) , secretHash : _secretHashForL2MessageConsumption , deadline : _deadline , fee : _fee }) ; bytes32 entryKey = inbox . cancelL2Message (message ,

address (this)) ; // release the funds to msg.sender (since the content hash (& entry key) is derived by hashing the caller, // we confirm that msg.sender is same as _canceller supplied when creating the message) underlying . transfer (msg . sender , _amount) ; return entryKey ; } Source code: l1-contracts/test/portals/TokenPortal.sol#L104-L184 The example above ensure that the user can cancel their message if it is underpriced.

Designated caller

Designating a caller grants the ability to specify who should be able to call a function that consumes a message. This is useful for ordering of batched messages.

When performing multiple cross-chain calls in one action it is important to consider the order of the calls. Say for example, that you want to perform a uniswap trade on L1 because you are a whale and slippage on L2 is too damn high.

You would practically, withdraw funds from the rollup, swap them on L1, and then deposit the swapped funds back into the rollup. This is a fairly simple process, but it requires that the calls are done in the correct order. For one, if the swap is called before the funds are withdrawn, the swap will fail. And if the deposit is called before the swap, the funds might get lost!

As message boxes only will allow the recipient portal to consume the message, we can use this to our advantage to ensure that the calls are done in the correct order. Say that we include a designated "caller" in the messages, and that the portal contract checks that the caller matches the designated caller or designated is address(0) (anyone can call). When the message are to be consumed on L1, it can compute the message as seen below:

bytes

memory message = abi . encodeWithSignature ("withdraw(uint256,address,address)" , _amount , _to , _withCaller ? msg . sender :

address (0)) ; This way, the message can be consumed by the portal contract, but only if the caller is the designated caller. By being a bit clever when specifying the designated caller, we can ensure that the calls are done in the correct order. For the Uniswap example, say that we have token portals implemented as we have done throughout this page, and a Uniswap portal implementing the designated caller.

We require that the Uniswap portal is the caller of the withdrawal, and that the uniswap portal implementation is executing the withdrawal before the swap. The order of execution can be constrained in the contract. Since all of the messages are emitted to L1 in the same transaction, we can leverage transaction atomicity to ensure success of failure of all messages.

Note, that crossing the L1/L2 chasm is asynchronous, so there could be a situation where the user has burned their assets on L2 but the swap fails on L1! This could be due to major price movements or the like. In such a case, the user could be stuck with funds on L1 that they cannot get back to L2 unless the portal contract implements a way to properly handle such errors.

caution Designated callers are enforced at the contract level for contracts that are not the rollup itself, and should not be trusted to implement the contract correctly. The user should always be aware that it is possible for the developer to implement something that looks like designated caller without providing the abilities to the user.

Examples of portals

- Token bridge (Portal contract built for L1 -> L2, i.e., a non-native L2 asset)*Portal contract
-
    - Aztec contract Edit this page

Previous How to deploy a contract with a PortalNext Slow Updates Tree