Transfer USDC with Data

USDC is a digital dollar backed 100% and is always redeemable 1:1 for US dollars. Thestablecoin is issued byCircle on multiple blockchain platforms.

This guide will first explain how Chainlink CCIP enables native USDC transfers under the hood by leveragingCircle's Cross-Chain Transfer Protocol (CCTP) . Then, you will learn how to use Chainlink CCIP to transfer USDC and arbitrary data from a smart contract onAvalanche Fujito a smart contract onPolygon Mumbai.Note: In addition to programmable token transfers, you can also use CCIP to transfer USDC tokens without data. Check theMainnets andTestnets configuration pages to learn on which blockchains CCIP supports USDC transfers.

Architecture

Fundamentally thearchitecture of CCIP andAPI are unchanged:

- The sender has to interact with the CCIP router to initiate a cross-chain transaction, similar to the process for any other token transfers. See theTransfer Tokens guide to learn more.
- The process uses the same onchain components including the Router, OnRamp, Commit Store, OffRamp, and Token Pool.
- The process uses the same offchain components including the Committing DON, Executing DON, and the Risk Management Network.
- USDC transfers also benefit from CCIP additional security provided by theRisk Management Network .

The diagram below shows that the USDC token pools and Executing DON handle the integration with Circle's contracts and offchain CCTP Attestation API. As with any other supported ERC-20 token, USDC has a linked token pool on each supported blockchain to facilitate OnRamp and OffRamp operations. To learn more about these components, read thearchitecture page .

The following describes the operational process:

1. On the source blockchain:1. When the sender initiates a transfer of USDC, the USDC token pool interacts with CCTP's contract to burn USDC tokens and specifies the USDC token pool address on the destination blockchain as the authorized caller to mint them.
2. CCTP burns the specified USDC tokens and emits an associated CCTP event.
3. Offchain:1. Circle attestation service listens to CCTP events on the source blockchain.
4. CCIPExecuting DON listens to relevant CCTP events on the source blockchain. When it captures such an event, it calls the Circle Attestation service API to request an attestation. An attestation is a signed authorization to mint the specified amount of USDC on the destination blockchain.
5. On the destination blockchain:1. TheExecuting DON provides the attestation to theOffRamp contract .
6. The OffRamp contract calls the USDC token pool with the USDC amount to be minted, the Receiver address, and the Circle attestation.
7. The USDC token pool calls the CCTP contract. The CCTP contract verifies the attestation signature before minting the specified USDC amount into the Receiver.
8. If there is data in the CCIP message and the Receiver is not an EOA, then the OffRamp contract transmits the CCIP message via theRouter contract to the Receiver.

Example

In this tutorial, you will send astringtext and USDC tokens from a smart contract onAvalanche Fujito a smart contract onPolygon Mumbai. You will pay CCIP fees in LINK. For simplicity, we will use the same contract example as theTransfer Tokens with Data tutorial but for production code, we recommend to apply defensive coding (read theTransfer Tokens With Data - Defensive Example tutorial to learn more).

Before you begin

1. You should understand how to write, compile, deploy, and fund a smart contract. If you need to brush up on the basics, read thistutorial , which will guide you through using theSolidity programming language , interacting with theMetaMask wallet and working within theRemix Development Environment .
2. Your account must have some AVAX and LINK tokens onAvalanche Fujiand MATIC tokens onPolygon Mumbai. You can use theChainlink faucet to acquire testnet tokens.
3. Check theSupported Networks page to confirm that USDC are supported for your lane. In this example, you will transfer tokens fromAvalanche FujitoPolygon Mumbaiso check the list of supported tokenshere .
4. Use theCircle faucet to acquire USDC tokens onAvalanche Fuji.
5. Learn how tofund your contract . This guide shows how to fund your contract in LINK, but you can use the same guide for funding your contract with any ERC-20 tokens as long as they appear in the list of tokens in MetaMask.

Tutorial

// SPDX-License-Identifier: MITpragmasolidity0.8.19;import{IRouterClient}from"@chainlink/contracts-ccip/src/v0.8/ccip/interfaces/IRouterClient.sol";import{OwnerIsCreator}from"@chainlink/contracts-ccip/src/v0.8/shared/access/OwnerIsCreator.sol";import{Client}from"@chainlink/contracts-ccip/src/v0.8/ccip/libraries/Client.sol";import{CCIPReceiver}from"@chainlink/contracts-ccip/src/v0.8/ccip/applications/CCIPReceiver.sol";import{IERC20}from"@chainlink/contracts-ccip/src/v0.8/vendor/openzeppelin-solidity/v4.8.3/contracts/token/ERC20/IERC20.sol";/ *** THIS IS AN EXAMPLE CONTRACT THAT USES HARDCODED VALUES FOR CLARITY. * THIS IS AN EXAMPLE CONTRACT THAT USES UN-AUDITED CODE. * DO NOT USE THIS CODE IN PRODUCTION. *//// @title - A simple messenger contract for transferring/receiving tokens and data across chains.contractProgrammableTokenTransfersisCCIPReceiver,OwnerIsCreator{// Custom errors to provide more descriptive revert messages.errorNotEnoughBalance(uint256currentBalance,uint256calculatedFees);// Used to make sure contract has enough balance to cover the fees.errorNothingToWithdraw();// Used when trying to withdraw Ether but there's nothing to withdraw.errorFailedToWithdrawEth(addressowner,addresstarget,uint256value);// Used when the withdrawal of Ether fails.errorDestinationChainNotAllowed(uint64destinationChainSelector);// Used when the destination chain has not been allowlisted by the contract owner.errorSourceChainNotAllowed(uint64sourceChainSelector);// Used when the source chain has not been allowlisted by the contract owner.errorSenderNotAllowed(addresssender);// Used when the sender has not been allowlisted by the contract owner.errorInvalidReceiverAddress();// Used when the receiver address is 0.// Event emitted when a message is sent to another chain.eventMessageSent(bytes32indexedmessageId,// The unique ID of the CCIP message.uint64indexeddestinationChainSelector,// The chain selector of the destination chain.addressreceiver,// The address of the receiver on the destination chain.stringtext,// The text being sent.addresstoken,// The token address that was transferred.uint256tokenAmount,// The token amount that was transferred.addressfeeToken,// the token address used to pay CCIP fees.uint256fees// The fees paid for sending the message.);// Event emitted when a message is received from another chain.eventMessageReceived(bytes32indexedmessageId,// The unique ID of the CCIP message.uint64indexedsourceChainSelector,// The chain selector of the source chain.addresssender,// The address of the sender from the source chain.stringtext,// The text that was received.addresstoken,// The token address that was transferred.uint256tokenAmount// The token amount that was transferred.);bytes32private_lastReceivedMessageId;// Store the last received messageId.addressprivate_lastReceivedTokenAddress;// Store the last received token address.uint256private_lastReceivedTokenAmount;// Store the last received amount.stringprivate_lastReceivedText;// Store the last received text.// Mapping to keep track of allowlisted destination chains.mapping(uint64=>bool)publicallowlistedDestinationChains;// Mapping to keep track of allowlisted source chains.mapping(uint64=>bool)publicallowlistedSourceChains;// Mapping to keep track of allowlisted senders.mapping(address=>bool)publicallowlistedSenders;IERC20private_linkToken;/// @notice Constructor initializes the contract with the router address./// @param *router The address of the router contract.*/// @param *_link The address of the link contract.*constructor(address_router,address_link)CCIPReceiver(_router) *{s_linkToken=IERC20(_link);}*/// @dev Modifier that checks if the chain with the given destinationChainSelector is allowlisted.*/// @param _destinationChainSelector The selector of the destination chain.*modifieronlyAllowlistedDestinationChain(uint64_destinationChainSelector) *{if(!allowlistedDestinationChains[_destinationChainSelector])revertDestinationChainNotAllowed(_destinationChainSelector);;}*/// @dev Modifier that checks the receiver address is not 0./// @param *receiver The receiver address.*modifiervalidateReceiver(address_receiver){if(_receiver==address(0))revertInvalidReceiverAddress();;}/// @dev Modifier that checks if the chain with the given sourceChainSelector is allowlisted and if the sender is allowlisted./// @param *sourceChainSelector The selector of the destination chain.*/// @param *_sender The address of the sender.*modifieronlyAllowlisted(uint64_sourceChainSelector,address_sender) *{if(!allowlistedSourceChains[_sourceChainSelector])revertSourceChainNotAllowed(_sourceChainSelector);if(!allowlistedSenders[_sender])revertSenderNotAllowed(_sender);;}*/// @dev Updates the allowlist status of a destination chain for transactions./// @notice This function can only be called by the owner./// @param _destinationChainSelector The selector of the destination chain to be updated./// @param allowed The allowlist status to be set for the destination chain.functionallowlistDestinationChain(uint64_destinationChainSelector,boolallowed)externalonlyOwner{allowlistedDestinationChains[_destinationChainSelector]=allowed;}/// @dev Updates the allowlist status of a source chain/// @notice This function can only be called by the owner./// @param _sourceChainSelector The selector of the source chain to be updated./// @param allowed The allowlist status to be set for the source chain.functionallowlistSourceChain(uint64_sourceChainSelector,boolallowed)externalonlyOwner{allowlistedSourceChains[_sourceChainSelector]=allowed;}/// @dev Updates the allowlist status of a sender for transactions./// @notice This function can only be called by the owner./// @param _sender The address of the sender to be updated./// @param allowed The allowlist status to be set for the sender.functionallowlistSender(address_sender,boolallowed)externalonlyOwner{allowlistedSenders[_sender]=allowed;}/// @notice Sends data and transfer tokens to receiver on the destination chain./// @notice Pay for fees in LINK./// @dev Assumes your contract has sufficient LINK to pay for CCIP fees./// @param _destinationChainSelector The identifier (aka selector) for the destination blockchain./// @param _receiver The address of the recipient on the destination blockchain./// @param _text The string data to be sent./// @param _token token address./// @param _amount token amount./// @return messageId The ID of the CCIP message that was sent.functionsendMessagePayLINK(uint64_destinationChainSelector,address_receiver,stringcalldata_text,address_token,uint256_amount)externalonlyOwneronlyAllowlistedDestinationCh {// Create an EVM2AnyMessage struct in memory with necessary information for sending a cross-chain message// address(linkToken) means fees are paid in LINKClient.EVM2AnyMessagememoryevm2AnyMessage=_buildCCIPMessage(_receiver,_text,_token,_amount,address(s_linkToken));// Initialize a router client instance to interact with cross-chain routerIRouterClient router=IRouterClient(this.getRouter());// Get the fee required to send the CCIP messageuint256fees=router.getFee(_destinationChainSelector,evm2AnyMessage);if(fees>s_linkToken.balanceOf(address(this)))revertNotEnoughBalance(s_linkToken.balanceOf(address(t approve the Router to transfer LINK tokens on contract's behalf. It will spend the fees in LINKs_linkToken.approve(address(router),fees);// approve the Router to spend tokens on contract's behalf. It will spend the amount of the given tokenIERC20(_token).approve(address(router),_amount);// Send the message through the router and store the returned message IDmessageId=router.ccipSend(_destinationChainSelector,evm2AnyMessage);// Emit an event with message detailsemitMessageSent(messageId,_destinationChainSelector,_receiver,_text,_token,_amount,address(s_linkToken),fees);// Return the message IDreturnmessageId;}/// @notice Sends data and transfer tokens to receiver on the destination chain./// @notice Pay for fees in native gas./// @dev Assumes your contract has sufficient native gas like ETH on Ethereum or MATIC on Polygon./// @param _destinationChainSelector The identifier (aka selector) for the destination blockchain./// @param _receiver The address of the recipient on the destination blockchain./// @param _text The string data to be sent./// @param _token token address./// @param _amount token amount./// @return messageId The ID of the CCIP message that was sent.functionsendMessagePayNative(uint64_destinationChainSelector,address_receiver,stringcalldata_text,address_token,uint256_amount)externalonlyOwneronlyAllowlistedDestinationC {// Create an EVM2AnyMessage struct in memory with necessary information for sending a cross-chain message// address(0) means fees are paid in native gasClient.EVM2AnyMessagememoryevm2AnyMessage=_buildCCIPMessage(_receiver,_text,_token,_amount,address(0));// Initialize a router client instance to interact with cross-chain routerIRouterClient router=IRouterClient(this.getRouter());// Get the fee required to send the CCIP

messageuint256fees=router.getFee(_destinationChainSelector,evm2AnyMessage);if(fees>address(this).balance)revertNotEnoughBalance(address(this).balance,fees);// approve the Router to spend tokens on contract's behalf. It will spend the amount of the given tokenIERC20(_token).approve(address(router),_amount);// Send the message through the router and store the returned message IDmessageId=router.ccipSend{value:fees}(_destinationChainSelector,evm2AnyMessage);// Emit an event with message detailsemitMessageSent(messageId,_destinationChainSelector,_receiver,_text,_token,_amount,address(0),fees);// Return the message IDreturnmessageId;}/ * @notice Returns the details of the last CCIP received message. * @dev This function retrieves the ID, text, token address, and token amount of the last received CCIP message. * @return messageId The ID of the last received CCIP message. * @return text The text of the last received CCIP message. * @return tokenAddress The address of the token in the last CCIP received message. * @return tokenAmount The amount of the token in the last CCIP received message. */functiongetLastReceivedMessageDetails()publicviewreturns(bytes32messageId,stringmemorytext,addresstokenAddress,uint256tokenAmount){return(s_lastReceivedMessageId,s_lastReceivedText,s_lastReceivedTokenAddress,s_lastReceivedTokenAmount);}/// handle a received messagefunction_ccipReceive(Client.Any2EVMMessagememoryany2EvmMessage)internaloverrideonlyAllowlisted(any2EvmMessage.sourceChainSelector,abi.decode(any2EvmMessage.sender,(address)))// Make sure source chain and sender are allowlisted{s_lastReceivedMessageId=any2EvmMessage.messageId;// fetch the messageIds_lastReceivedText=abi.decode(any2EvmMessage.data,(string));// abi-decoding of the sent text// Expect one token to be transferred at once, but you can transfer several tokens.s_lastReceivedTokenAddress=any2EvmMessage.destTokenAmounts[0].token;s_lastReceivedTokenAmount=any2EvmMessage.destTokenAmounts[0].amount;emitMessageReceived(any2EvmMessage.messageId,any2EvmMessage.sourceChainSelector,// fetch the source chain identifier (aka selector)abi.decode(any2EvmMessage.sender,(address)),// abi-decoding of the sender address,abi.decode(any2EvmMessage.data,(string)),any2EvmMessage.destTokenAmounts[0].token,any2EvmMessage.destTokenAmounts[0].amount);}/// @notice Construct a CCIP message./// @dev This function will create an EVM2AnyMessage struct with all the necessary information for programmable tokens transfer./// @param _receiver The address of the receiver./// @param _text The string data to be sent./// @param _token The token to be transferred./// @param _amount The amount of the token to be transferred./// @param _feeTokenAddress The address of the token used for fees. Set address(0) for native gas./// @return Client.EVM2AnyMessage Returns an EVM2AnyMessage struct which contains information for sending a CCIP message.function_buildCCIPMessage(address_receiver,stringcalldata_text,address_token,uint256_amount,address_feeTokenAddress)privatepurereturns(Client.EVM2AnyMessagememory){// Set the token amountsClient.EVMTokenAmount[]memorytokenAmounts=newClient.EVMTokenAmount;tokenAmounts[0]=Client.EVMTokenAmount({token:_token,amount:_amount});// Create an EVM2AnyMessage struct in memory with necessary information for sending a cross-chain messagereturnClient.EVM2AnyMessage({receiver:abi.encode(_receiver),// ABI-encoded receiver addressdata:abi.encode(_text),// ABI-encoded stringtokenAmounts:tokenAmounts,// The amount and type of token being transferredextraArgs:Client._argsToBytes(// Additional arguments, setting gas limitClient.EVMExtraArgsV1({gasLimit:200_000})),// Set the feeToken to a feeTokenAddress, indicating specific asset will be used for feesfeeToken:_feeTokenAddress});}/// @notice Fallback function to allow the contract to receive Ether./// @dev This function has no function body, making it a default function for receiving Ether./// It is automatically called when Ether is sent to the contract without any data.receive()externalpayable{}/// @notice Allows the contract owner to withdraw the entire balance of Ether from the contract./// @dev This function reverts if there are no funds to withdraw or if the transfer fails./// It should only be callable by the owner of the contract./// @param _beneficiary The address to which the Ether should be sent.functionwithdraw(address_beneficiary)publiconlyOwner{// Retrieve the balance of this contractuint256amount=address(this).balance;// Revert if there is nothing to withdrawif(amount==0)revertNothingToWithdraw();// Attempt to send the funds, capturing the success status and discarding any return data(boolsent,)=_beneficiary.call{value:amount}("");// Revert if the send failed, with information about the attempted transferif(!sent)revertFailedToWithdrawEth(msg.sender,_beneficiary,amount);}/// @notice Allows the owner of the contract to withdraw all tokens of a specific ERC20 token./// @dev This function reverts with a 'NothingToWithdraw' error if there are no tokens to withdraw./// @param _beneficiary The address to which the tokens will be sent./// @param _token The contract address of the ERC20 token to be withdrawn.functionwithdrawToken(address_beneficiary,address_token)publiconlyOwner{// Retrieve the balance of this contractuint256amount=IERC20(_token).balanceOf(address(this));// Revert if there is nothing to withdrawif(amount==0)revertNothingToWithdraw();IERC20(_token).transfer(_beneficiary,amount);}}** Open in Remix What is Remix?

Deploy your contracts

To use this contract:

1. Open the contract in Remix.
2. Compile your contract.
3. Deploy, fund your sender contract onAvalanche Fujiand enable sending messages toPolygon Mumbai:
4. Open MetaMask and select the networkAvalanche Fuji.
5. In Remix IDE, click onDeploy & Run Transactionsand selectInjected Provider - MetaMaskfrom the environment list. Remix will then interact with your MetaMask wallet to communicate withAvalanche Fuji.
6. Fill in your blockchain's router and LINK contract addresses. The router address can be found on thesupported networks page and the LINK contract address on theLINK token contracts page . ForAvalanche Fuji, the router address is0xf694e193200268f9a4868e4aa017a0118c9a8177and the LINK contract address is0x0b9d5D9136855f6FEc3c0993feE6E9CE8a297846.
7. Click thetransactbutton. After you confirm the transaction, the contract address appears on theDeployed Contractslist. Note your contract address.
8. Open MetaMask and fund your contract with USDC tokens. You can transfer1USDCto your contract.
9. Fund your contract with LINK tokens. You can transfer0.5LINKto your contract. In this example, LINK is used to pay the CCIP fees.
10. Enable your contract to send CCIP messages toPolygon Mumbai:1. In Remix IDE, underDeploy & Run Transactions, open the list of transactions of your smart contract deployed onAvalanche Fuji.
11. Call theallowlistDestinationChainwith12532609583862916517as the destination chain selector, andtrueas allowed. Each chain selector is found on thesupported networks page .
12. Deploy your receiver contract onPolygon Mumbaiand enable receiving messages from the sender contract:
13. Open MetaMask and select the networkPolygon Mumbai.
14. In Remix IDE, underDeploy & Run Transactions, make sure the environment is stillInjected Provider - MetaMask.
15. Fill in your blockchain's router and LINK contract addresses. The router address can be found on thesupported networks page and the LINK contract address on theLINK token contracts page . ForPolygon Mumbai, the router address is0x1035CabC275068e0F4b745A29CEDf38E13aF41b1and the LINK contract address is0x326C977E6efc84E512bB9C30f76E30c160eD06FB.
16. Click thetransactbutton. After you confirm the transaction, the contract address appears on theDeployed Contractslist. Note your contract address.
17. Enable your contract to receive CCIP messages fromAvalanche Fuji:1. In Remix IDE, underDeploy & Run Transactions, open the list of transactions of your smart contract deployed onPolygon Mumbai.
18. Call theallowlistSourceChainwith14767482510784806043as the source chain selector, andtrueas allowed. Each chain selector is found on thesupported networks page .
19. Enable your contract to receive CCIP messages from the contract that you deployed onAvalanche Fuji:1. In Remix IDE, underDeploy & Run Transactions, open the list of transactions of your smart contract deployed onPolygon Mumbai.
20. Call theallowlistSenderwith the contract address of the contract that you deployed onAvalanche Fuji, andtrueas allowed.

At this point, you have onesendercontract onAvalanche Fujiand onereceivercontract onPolygon Mumbai. As security measures, you enabled the sender contract to send CCIP messages toPolygon Mumbaiand the receiver contract to receive CCIP messages from the sender andAvalanche Fuji.

Transfer and Receive tokens and data and pay in LINK

You will transfer1 USDCand a text. The CCIP fees for using CCIP will be paid in LINK.

1. Send a string data with tokens fromAvalanche Fuji:
2. Open MetaMask and select the networkAvalanche Fuji.
3. In Remix IDE, underDeploy & Run Transactions, open the list of transactions of your smart contract deployed onAvalanche Fuji.
4. Fill in the arguments of thesendMessagePayLINKfunction:

ArgumentValue and Description_destinationChainSelector12532609583862916517CCIP Chain identifier of the destination blockchain (Polygon Mumbaiin this example). You can find each chain selector on thesupported networks page ._receiverYour receiver contract address atPolygon Mumbai.The destination contract address._textHello World!Anystring_token0x5425890298aed601595a70AB815c96711a31Bc65TheUSDCcontract address at the source chain (Avalanche Fujiin this example). You can find all the addresses for each supported blockchain on thesupported networks page ._amount1000000The token amount (1 USDC). 4. Click ontransactand confirm the transaction on MetaMask. 5. After the transaction is successful, record the transaction hash. Here is anexample of a transaction onAvalanche Fuji.

note

During gas price spikes, your transaction might fail, requiring more than0.5 LINKto proceed. If your transaction fails, fund your contract with moreLINKtokens and try again. 2. Open theCCIP explorer and search your cross-chain transaction using the transaction hash. 3. The CCIP transaction is completed once the status is marked as "Success". In this example, the CCIP message ID is0x083a235e1b723b9a304cb50d837ad704bfc824a0ef4b8174560fcd85d7af31fc. 4. Check the receiver contract on the destination chain:

1. Open MetaMask and select the networkPolygon Mumbai.
2. In Remix IDE, underDeploy & Run Transactions, open the list of transactions of your smart contract deployed onPolygon Mumbai.
3. Call thegetLastReceivedMessageDetailsfunction.
4. Notice the received messageId is0x083a235e1b723b9a304cb50d837ad704bfc824a0ef4b8174560fcd85d7af31fc, the received text isHello World!, the token address is0x9999f7Fea5938fD3b1E26A12c3f2fb024e194f97(USDC token address onPolygon Mumbai) and the token amount is 1000000 (1 USDC).

Note: These example contracts are designed to work bi-directionally. You can use them as an exercise to transfer tokens with data fromAvalanche FujitoPolygon Mumbaiand fromPolygon Mumbaiback toAvalanche Fuji. Always ensure the sender contract on the source chain is funded with enough fee tokens.