

Getting Started with Chainlink Data Streams

Mainnet Access

Chainlink Data Streams is available on Arbitrum Mainnet and Arbitrum Sepolia.

Talk to an expert

[Contact us](#) to talk to an expert about integrating Chainlink Data Streams with your applications.

This guide shows you how to read data from a Data Stream, validate the answer, and store the answer onchain. This example uses [@chainlink Automation Log Trigger](#) to check for events that require data. For this example, the log trigger comes from a simple emitter contract. Chainlink Automation then uses `StreamsLookup` to retrieve a signed report from the Data Streams Engine, return the data in a callback, and run the `performUpkeep` function on your registered upkeep contract. The `performUpkeep` function calls the `verify` function on the verifier contract.

Disclaimer

This guide represents an example of using a Chainlink product or service and is provided to help you understand how to interact with Chainlink's systems and services so that you can integrate them into your own. This template is provided "AS IS" and "AS AVAILABLE" without warranties of any kind, has not been audited, and may be missing key checks or error handling to make the usage of the product more clear. Do not use the code in this example in a production environment without completing your own audits and application of best practices. Neither Chainlink Labs, the Chainlink Foundation, nor Chainlink node operators are responsible for unintended outputs that are generated due to errors in code.

Before you begin

- If you are new to smart contract development, learn how to [Deploy Your First Smart Contract](#) so you are familiar with the tools that are necessary for this guide.* The [Solidity](#) programming language
- The [MetaMask](#) wallet
- The [Remix](#) development environment
- Acquire testnet funds. This guide requires testnet ETH and LINK on Arbitrum Sepolia.* Use the [Arbitrum Bridge](#) to transfer testnet ETH from Ethereum Sepolia to Arbitrum Sepolia. Testnet ETH on Ethereum Sepolia is available at one of [several faucets](#).
- Testnet LINK is available for Arbitrum Sepolia at [faucets.chain.link](#).
- Learn how to [Fund your contract with LINK](#).

Deploy the Chainlink Automation upkeep contract

Deploy an upkeep contract that is enabled to retrieve data from Data Streams. For this example, you will read from the ETH/USD stream with ID `0x00027bbaff688c906a3e20a34fe951715d1018d262a5b66e38eda027a674cd1b0n` Arbitrum Sepolia. See the [Stream Identifiers](#) page for a complete list of available assets, IDs, and verifier proxy addresses.

1. [Open the StreamsUpkeep.sol](#) contract in Remix.

[Open in Remix](#) [What is Remix?](#) 2. Select the 0.8.16 Solidity compiler and the `StreamsUpkeep.sol` contract in the Solidity Compiler tab. 3. Compile the contract. You can ignore the warning messages for this example. 4. Open MetaMask and set the network to Arbitrum Sepolia. If you need to add Arbitrum Sepolia to your wallet, you can find the chain ID and the LINK token contract address on the [LINK Token Contracts](#) page.

- [Arbitrum Sepolia testnet and LINK token contract](#)
- On the Deploy & Run Transaction tab in Remix, select Injected Provider - MetaMask in the Environment list. Remix will use the MetaMask wallet to communicate with Arbitrum Sepolia.
- In the Contract section, select the `StreamsUpkeep` contract and fill in the verifier proxy address corresponding to the stream you want to read from. You can find this address on the [Stream IDs](#) page. The verifier proxy address for the ETH/USD stream on Arbitrum Sepolia is `0x2ff010DEbC1297f19579B4246cad07bd24F2488A`.
- Click the Deploy button to deploy the contract. MetaMask prompts you to confirm the transaction. Check the transaction details to ensure you deploy the contract to Arbitrum Sepolia.
- After you confirm the transaction, the contract address appears under the Deployed Contracts list in Remix. Save this contract address for later.

Deploy the emitter contract

This contract emits logs that trigger the upkeep. This code can be part of your dApp. For example, you might emit log triggers when your users initiate a trade or other action requiring data retrieval. For this Getting Started guide, use a very simple emitter so you can test the upkeep and data retrieval.

1. [Open the LogEmitter.sol](#) contract in Remix.

[Open in Remix](#) [What is Remix?](#) 2. Under the Solidity Compiler tab, select the 0.8.19 Solidity compiler and click the Compile `LogEmitter.sol` button to compile the contract. 3. Open MetaMask and make sure the network is still set to Arbitrum Sepolia. 4. On the Deploy & Run Transaction tab in Remix, ensure the Environment is still set to Injected Provider - MetaMask. 5. Click the Deploy button to deploy the contract. MetaMask prompts you to confirm the transaction. Check the transaction details to ensure you deploy the contract to Arbitrum Sepolia. 6. After you confirm the transaction, the contract address appears in the Deployed Contracts list. Save this contract address for later.

Register the upkeep

Register a new Log trigger upkeep. See [Automation Log Triggers](#) to learn more about how to register Log Trigger upkeep.

1. Go to the [Chainlink Automation UI](#) for Arbitrum Sepolia and connect your browser wallet.
2. Click Register new Upkeep.
3. Select the Log trigger upkeep type and click Next.
4. Specify the upkeep contract address you saved earlier as the Contract to automate. In this example, you can ignore the warning about the Automation compatible contract verification. Click Next.
5. Specify the emitter contract address that you saved earlier. This tells Chainlink Automation what contracts to watch for log triggers. Then click Next.
6. Provide the ABI if the contract is not validated. To find the ABI of your contract in Remix, navigate to the Solidity Compiler tab. Then, copy the ABI to your clipboard using the button at the bottom of the panel.
7. Select the `LogEvent` as the triggering event in the Emitted log dropdown. Log index topic filters are optional filters to narrow the logs you want to trigger your upkeep. For this example, leave the field blank. Click Next.
8. Specify a name for the upkeep.
9. Specify a starting balance of 1 testnet LINK for this example. You can retrieve unused LINK later.
10. Leave the Check data value and other fields blank for now, and click Register Upkeep. MetaMask prompts you to confirm the transaction. Wait for the transaction to complete.

Fund the upkeep contract

In this example, the upkeep contract pays for onchain verification of reports from Data Streams. The Automation subscription does not cover the cost.

Open MetaMask and send 1 testnet LINK on Arbitrum Sepolia to the upkeep contract address you saved earlier.

Emit a log

Now, you can use your emitter contract to emit a log and initiate the upkeep, which retrieves data for the specified Data Streams asset ID.

1. In Remix, on the Deploy & Run Transaction tab, expand your emitter contract under the Deployed Contracts section.
2. Click the `emitLog` button to call the function and emit a log. MetaMask prompts you to accept the transaction.

After the transaction is complete, the log is emitted, and the upkeep is triggered. You can find the upkeep transaction hash in the [@chainlink Automation UI](#). Check to make sure the transaction is successful.

View the retrieved price

The retrieved price is stored as a variable in the contract and emitted in the logs.

1. On the Deploy & Run Transaction tab in Remix, expand the details of your upkeep contract in the Deployed Contracts section.
2. Click the `last_retrieved_price` getter function to view the retrieved price. The answer on the ETH/USD stream uses 18 decimal places, so an answer of `2484121000000000000` indicates an ETH/USD price of 2484.121. Each stream uses a different number of decimal places for answers. See the [Stream IDs](#) page for more information.

Alternatively, you can view the price emitted in the logs for your upkeep transaction. You can find the upkeep transaction hash at [@chainlink Automation UI](#) and view the transaction logs in the [Arbitrum Sepolia explorer](#).

Examine the code

The example code you deployed has all the interfaces and functions required to work with Chainlink Automation as an upkeep contract. It follows a similar flow to the trading flow in the [Architecture](#) documentation but uses a basic log emitter to simulate the client contract that would initiate aStreamsLookup. After the contract receives and verifies the report,performUpkeepstores the price from the report in thelast_retrieved_priceand emits aPriceUpdatelog message with the price. You could modify this to use the data in a way that works for your specific use case and application.

The code example usesrevertwithStreamsLookupto convey call information about what streams to retrieve. See the [EIP-3668 rationale](#) for more information about how to userevertin this way.

```
// SPDX-License-Identifier:
MIT
pragma solidity ^0.8.16;
import {Common} from "@chainlink/contracts/src/v0.8/libraries/Common.sol";
import {StreamsLookupCompatibleInterface} from "@chainlink/contracts/src/v0.8/automation/interfaces/feeds/interfaces/IRewardManager.sol";
import {IVerifierFeeManager} from "@chainlink/contracts/src/v0.8/feeds/interfaces/IVerifierFeeManager.sol";
import {IERC20} from "@chainlink/contracts/src/v0.8/vendor/openzeppelin-solidity/v4.8.0/contracts/interfaces/IERC20.sol";

* THIS IS AN EXAMPLE CONTRACT THAT USES UN-AUDITED CODE FOR DEMONSTRATION PURPOSES. * DO NOT USE THIS CODE IN PRODUCTION.

// Custom interfaces for IVerifierProxy and IFeeManager
interface IVerifierProxy {
    function verify(bytes calldata payload, bytes calldata parameterPayload) external payable returns (bytes memory verifierResponse);
}

function _feeManager() external view returns (IFeeManager);

// The feed ID the report has data for
uint32 validFromTimestamp; // Earliest timestamp for which price is applicable
uint32 observationsTimestamp; // Latest timestamp for which price is applicable
uint192 nativeFee; // Base cost to validate a transaction using the report, denominated in the chain's native token (WETH/ETH)
uint192 linkFee; // Base cost to validate a transaction using the report, denominated in LINK
uint32 expiresAt; // Latest timestamp where the report can be verified onchain
int192 price; // DON consensus median price, carried to 8 decimal places
struct PremiumReport {
    bytes32 feedId; // The feed ID the report has data for
    uint32 validFromTimestamp; // Earliest timestamp for which price is applicable
    uint32 observationsTimestamp; // Latest timestamp for which price is applicable
    uint192 nativeFee; // Base cost to validate a transaction using the report, denominated in the chain's native token (WETH/ETH)
    uint192 linkFee; // Base cost to validate a transaction using the report, denominated in LINK
    uint32 expiresAt; // Latest timestamp where the report can be verified onchain
    int192 price; // DON consensus median price, carried to 8 decimal places
    int192 bid; // Simulated price impact of a buy order up to the X% depth of liquidity utilisation
    int192 ask; // Simulated price impact of a sell order up to the X% depth of liquidity utilisation
}

struct Quote {
    address quoteAddress;
    event PriceUpdate(int192 indexed price);
}

IVerifierProxy public verifier;
address public FEE_ADDRESS;
string public constant DATASTREAMS_FEEDLABEL = "feed";

This example reads the ID for the basic ETH/USD price report on Arbitrum Sepolia. Find a complete list of IDs at https://docs.chain.link/data-streams/stream-idsstring[publicfeedIds=["0x00027bbaff688c906a3e20a34fe951715d1018d262a5b66e38eda027a674cd1b"]];
constructor(address _verifier) {
    verifier = IVerifierProxy(_verifier);
}

// This function uses revert to convey call information. See https://eips.ethereum.org/EIPS/eip-3668#rationale for details.
function checkLog(Log calldata log, bytes memory) external returns (bool upkeepNeeded, bytes memory performData) {
    (revertStreamsLookup(DATASTREAMS_FEEDLABEL, feedIds, DATASTREAMS_QUERYLABEL, log.timestamp, ""));
    // The Data Streams report bytes is passed here. extraData is context data from feed lookup process. Your contract may include logic to further process this data. This method is intended only to be simulated offchain by Automation. The data returned will then be passed by Automation into performUpkeep.
    function checkCallback(bytes calldata values, bytes calldata extraData) external pure returns (bool, bytes memory) {
        return (true, abi.encode(values, extraData));
    }
    // function will be performed onchain.
    function performUpkeep(bytes calldata performData) external {
        // Decode the performData bytes passed in by CL Automation. This contains the data returned by your implementation in checkCallback().
        (bytes[] memory signedReports, bytes memory extraData) = abi.decode(performData, (bytes[], bytes));
        bytes memory unverifiedReport = signedReports[0];
        (bytes32[3] reportContextData, bytes memory reportData) = abi.decode(unverifiedReport, (bytes32[3], bytes));
        // Report verification
        fees IFeeManager feeManager = IFeeManager(address(verifier._feeManager()));
        IRewardManager rewardManager = IRewardManager(address(feeManager._rewardManager()));
        address feeTokenAddress = feeManager._linkAddress();
        (Common.Asset memory fee,) = feeManager.getFeeAndReward(address(this), reportData, feeTokenAddress);
        // Approve rewardManager to spend this contract's balance in fees
        IERC20(feeTokenAddress).approve(address(rewardManager), fee.amount);
        // Verify the report
        bytes memory verifiedReportData = verifier.verify(unverifiedReport, abi.encode(feeTokenAddress));
        // Decode verified report data into BasicReport struct
        BasicReport memory verifiedReport = abi.decode(verifiedReportData, (BasicReport));
        // Log price from report
        emit PriceUpdate(verifiedReport.price);
        // Store the price from the report
        last_retrieved_price = verifiedReport.price;
    }
}

fallback() external payable {}
```

[Open in Remix](#) [What is Remix?](#)