

Using the StreamsLookup error handler

Data Streams Mainnet Access

Chainlink Data Streams is available on Arbitrum Mainnet and Arbitrum Sepolia. [Contact us](#) to talk to an expert about integrating Chainlink Data Streams with your applications.

The Chainlink Automation StreamsLookup error handler provides insight into potential errors or edge cases in StreamsLookup upkeeps. The table below outlines a range of error codes and the behavior associated with the codes. Use the `checkErrorHandlerFunction` to specify how you want to respond to the error codes. `checkErrorHandler` is simulated offchain and determines what action for Automation to take onchain in `performUpkeep`.

Developer responsibility

Developers implementing Chainlink products are solely responsible for maintaining the security and user experience of their applications. Developers must monitor and mitigate any potential application code risks that may, among other things, result in unanticipated application behavior, including by instituting requisite [risk mitigation processes](#) including, but not limited to, data quality checks, circuit breakers, and appropriate contingency logic for their use case.

Error handler

When Automation detects an event, it runs the `checkLog` function, which includes [StreamsLookup](#) revert custom error. The `StreamsLookup` revert enables your upkeep to fetch a report from Data Streams. If reports are fetched successfully, the `checkCallback` function is evaluated offchain. Otherwise, the `checkErrorHandler` function is evaluated offchain to determine what Automation should do next. Both of these functions have the same output types (`bool` `upkeepNeeded`, `bytes` `memory` `performData`), which Automation uses to run `performUpkeep` onchain. The [example code](#) also shows each function outlined in the diagram below:

If the Automation network fails to get the requested reports, an error code is sent to the `checkErrorHandlerFunction` in your contract. If your contract doesn't have the `checkErrorHandlerFunction`, nothing will happen. If your contract has the `checkErrorHandlerFunction`, it is evaluated offchain to determine what to do next. For example, you could intercept or ignore certain errors and decide not to run `performUpkeep` in those cases, in order to save time and gas. For other errors, you can execute an alternative path within `performUpkeep`, and the upkeep runs the custom logic you define in `yourperformUpkeepFunction` to handle those errors.

1. Add the `checkErrorHandler` function in your contract to specify how you want to handle `error codes`. For example, you could decide to ignore any codes related to bad requests or incorrect input, without running `performUpkeep` on chain:

```
function checkErrorHandler(uint errorCode, bytes callData, extraData) external returns (bool upkeepNeeded, bytes memory performData) { // Add custom logic to handle errors of chain
    (bool _upkeepNeeded, bool success, bool isError, if (errorCode == 808400) { // Handle bad request error code of chain _upkeepNeeded = false; } else { // logic to handle other
    returns) return (_upkeepNeeded, abi.encode(isError, abi.encode(errorCode, extraData, success))); } 2. Define custom logic for the alternative path within performUpkeep, to handle any error codes you did
    not intercept of chain checkErrorHandler;
```

```
function will be performed on-chainfunctionperformUpkeep(bytescalldataperformData)external{// Decode incoming performData(booleisError,bytespayload)=abi.decode(performData)// Unpacking the
errorCode from checkErrorHandlerif(isError){(uinterrorCode,bytesmemoryextraData,boolreportSuccess)=abi.decode(payload,(uint,bytes,bool))// Define custom logic here to handle error codes
onchain);}else{// Otherwise unpacking info from checkCallback(bytes[]memorysignedReports,bytesmemoryextraData,boolreportSuccess)=abi.decode(payload,(bytes[],bytes,bool));if(reportSuccess)
{bytesmemoryreport=signedReports[0];(bytesmemoryreportData)=abi.decode(report,(bytes32[3],bytes))// Logic to verify and decode reportelse{// Logic in case reports were not pulled successfully}}
```

Testing checkErrorHandler

checkErrorHandler is simulated offchain. When `upkeepNeeded` returns true, Automation runs `performUpkeep` onchain using the `performData` from `checkErrorHandler`. If the `checkErrorHandler` function itself reverts, `performUpkeep` does not run.

If you need to force errors in `StreamsLookup` while testing, you can try the following methods:

- Specifying an incorrect feed ID to force error code 808400 (ErrCodeStreamsBadRequest)
- Specifying a future timestamp to force error code 808206 (where partial content is received) for both singlefeedIDand bulkfeedID requests
- Specifying old timestamps for reports not available anymore yields either error code 808504 (no response) or 808600 (bad response), depending on which service calls the timeout request

If your [StreamsLookup revert](#) function is defined incorrectly in your smart contracts, the nodes will not be able to decode it.

Error codes

Error codeRetriesPossible cause of errorNo errorErrCodeStreamsBadRequest: 808400NoUser requested 0 feedsUser error, incorrect parameter inputIssue with encoding http url (bad characters)ErrCodeStreamsUnauthorized: 808401NoKey access issue or incorrect feedID808206Log trigger - after retries; Conditional immediatelyRequested m reports but only received n (partial)8085XX (e.g. 808500)Log trigger - after retries; Conditional immediatelyNo responseErrCodeStreamsBadResponse: 808600NoError in reading body of returned response, but service is upErrCodeStreamsTimeout: 808601NoNo valid report is received for 10 secondsErrCodeStreamsUnknownError: 808700NoUnknown

Example code

This example code includes `therevert`, `StreamsLookup`, `checkCallback`, `checkErrorHandler` and `performUpkeep` functions. The full code example is available [here](#).

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.16; import('StreamsLookupCompatibleInterface.sol') from '@chainlink/contracts/src/v0.8/automation/interfaces/StreamsLookupCompatibleInterface.sol'; import({ILogAutomation, Log}) from 'logs/interfaces/IRewardManager.sol'; import({VerifierFeeManager} from '@chainlink/contracts/src/v0.8/flo-
rewards/interfaces/IVerifierFeeManager.sol'); import({IERC20} from '@chainlink/contracts/src/v0.8/vendor/openzeppelin-
solidity/v4.8.0/contracts/interfaces/IERC20.sol'); import({Common} from '@chainlink/contracts/src/v0.8/libraries/Common.sol'); * THIS IS AN EXAMPLE CONTRACT THAT USES UN-AUDITED CODE. *
DO NOT USE THIS CODE IN PRODUCTION.
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////INTERFACES//////////////////////////////////////////////////////////////////////////////////////////////////////////////////
interface FeeManager {
    function getFeeAndReward(address subscriber, bytes memory report, address quoteAddress) external returns (uint256 fee, uint256 reward);
}
interface StreamsLookupCompatibleInterface {
    struct BasicReport {
        bytes32 feedId; // The feed ID the report has
        uint32 validFromTimestamp; // Earliest timestamp for which price is applicable
        uint32 observedPrice; // Latest timestamp for which price is applicable
        uint92 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 on-chain
        int192 price; // DON consensus median price, carried to 18 decimal places
    }
    struct PremiumReport {
        bytes32 feedId; // The Feed ID the report has
        uint32 validFromTimestamp; // Earliest timestamp for which price is applicable
        uint32 observedPrice; // Latest timestamp for which price is applicable
        uint92 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 on-chain
        int192 price; // DON consensus median price, carried to 18 decimal places
        sint192 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
        string data; // Data associated with the report
    }
    event PriceUpdate(int192 index, price);
    IVerifier proxyPublicVerifiers;
    address public FEE_ADDRESS;
    string public constant STRING_DATASTREAMS_FEEDLABEL = "0x00027baff68c906a3e20a34fe951715d1018d262a5b66638eda027a674cd1b";
    Ex. Basic ETH/USD price report; constructor(address _verifier) {
        verifier = IVerifierProxy(_verifier);
    }
    Arbitrum Sepolia: 0x2ff010debcc1297f19579b4246cad07bd24f2488a
    function checkLog(Log callData, log, bytes memory) external returns (bool upkeepNeeded, bytes memory performData) {
        revert(StreamsLookup(STRING_DATASTREAMS_FEEDLABEL, feedIds, STRING_DATASTREAMS_QUERYLABEL, log.timestamp, ""));
    }
    function checkCallback(bytes[] calldata values, bytes calldata extraData) bool {
        bool upkeepNeeded = true;
        bool success = true;
        bool isError = false;
        return (upkeepNeeded, abi.encode(isError, abi.encode(values, extraData, success)));
    }
    function checkErrorHandler(handler(ErrorCode error)) bool {
        bool upkeepNeeded = true;
        bool success = false;
        bool isError = true;
        Add custom logic to handle errors of chain here if (errorCode == 808400) { // Bad request error code
            upkeepNeeded = false;
        } else { // logic to handle other errors
            return (upkeepNeeded, abi.encode(isError, abi.encode(errorCode, extraData, success)));
        }
    }
    function performOnChainFunction(performData) bool {
        Decode incoming performData (bool isError, bytes memory payload) = abi.decode(performData, (bool, bytes));
        Unpacking the errorCode from checkErrorHandler(isError) (uint8 errorCode, bytes memory extraData, bool reportSuccess) = abi.decode(payload, (uint, bytes, bool));
        Custom logic to handle error codes onchain
        else { // Otherwise unpacking info from checkCallback(bytes[] memory signedReports, bytes memory extraData, bool reportSuccess) = abi.decode(payload, (bytes[], bytes, bool));
            if (reportSuccess) {
                bytes memory report = signedReports[0];
                bytes memory reportData = abi.decode(report, (bytes32[3], bytes));
                Billing IFeeManager feeManager = IFeeManager(address(verifier.s_feeManager()));
                IRewardManager rewardManager = IRewardManager(address(feeManager.i_rewardManager()));
                address feeTokenAddress = feeManager.i_linkAddress();
                (Common.AssetMemory fee,) = feeManager.getFeeAndReward(address(this), reportData, feeTokenAddress);
                IERC20(feeTokenAddress).approve(address(rewardManager), fee.amount);
                Verify the report by bytes memory verifiedReportData = verifier.verify(report, 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);
            } else { // Logic in case reports were not pulled successfully
                fallback(external payable f) Open in Remix What is Remix?
            }
        }
    }
}
```