

# Deposit Flow

This guide explains the deposit flow process for L2 deposit transactions, triggered by transactions or events on L1. In Optimism terminology, "deposit transaction " refers to any L2 transaction that is triggered by a transaction or event on L1.

The process is somewhat similar to the way [most networking stacks work \(opens in a new tab\)](#). Information is encapsulated in lower layer packets on the sending side and then retrieved and used by those layers on the receiving side while going up the stack to the receiving application.

## L1 Processing

1. An L1 entity, either a smart contract or an externally owned account (EOA), sends a deposit transaction [to L2CrossDomainMessenger \(opens in a new tab\)](#)
2. , using [sendMessage \(opens in a new tab\)](#)
3. .
4. This function accepts three parameters:
5.
  - `_target`
6.
  - , target address on L2.
7.
  - `_message`
8.
  - , the L2 transaction's calldata, formatted as per the [ABI \(opens in a new tab\)](#)
9.
  - of the target account.
10.
  - `_minGasLimit`
11.
  - , the minimum gas limit allowed for the transaction on L2. Note that this is a minimum
12.
  - and the actual amount provided on L2 may be higher (but never lower) than the specified gas limit.
13.
  - Note that the actual amount provided on L2 will be higher, because the portal contract on L2 needs to do some processing before submitting the call to `_target`
14.
  - .
15. The L1 cross domain messenger calls [its own\\_send function \(opens in a new tab\)](#)
16. .
17. It uses these parameters:
18.
  - `_to`
19.
  - , the destination address, is the messenger on the other side.
20.
  - In the case of deposits, this is always [0x420007 \(opens in a new tab\)](#)
21.
  - .
22.
  - `_gasLimit`
23.
  - , the gas limit.
24.
  - This value is calculated using [the baseGas function \(opens in a new tab\)](#)
25.
  - .
26.
  - `_value`
27.
  - , the ETH that is sent with the message.
28.
  - This amount is taken from the transaction value.
29.
  - `_data`
30.
  - , the calldata for the call on L2 that is needed to relay the message.
31.
  - This is an [ABI encoded \(opens in a new tab\)](#)
32.
  - call to [relayMessage \(opens in a new tab\)](#)
33.
  - .
34. [sendMessage \(opens in a new tab\)](#)
35. calls the portal's [depositTransaction function \(opens in a new tab\)](#)
36. .
37. Note that other contracts can also call [depositTransaction \(opens in a new tab\)](#)
38. directly.
39. However, doing so bypasses certain safeguards, so in most cases it's a bad idea.
40. [The depositTransaction function \(opens in a new tab\)](#)
41. runs a few sanity checks, and then emits a [TransactionDeposited \(opens in a new tab\)](#)
42. event.

## L2 Processing

1. The op-node
2. component [looks for TransactionDeposited events on L1 \(opens in a new tab\)](#)
3. .
4. If it sees any such events, it [parses \(opens in a new tab\)](#)
5. them.
6. Next, op-node
7. [converts \(opens in a new tab\)](#)
8. those TransactionDeposited
9. events into [deposit transactions \(opens in a new tab\)](#)
10. .
11. In most cases user deposit transactions call the [relayMessage \(opens in a new tab\)](#)
12. function of [L2CrossDomainMessenger \(opens in a new tab\)](#)
13. .
14. relayMessage
15. runs a few sanity checks and then, if everything is good, [calls the real target contract with the relayed calldata \(opens in a new tab\)](#)
16. .

## Denial of service (DoS) prevention

As with all other L1 transactions, the L1 costs of a deposit are borne by the transaction's originator. However, the L2 processing of the transaction is performed by the Optimism nodes. If there were no cost attached, an attacker could be able to submit a transaction that had high costs of run on L2, and that way perform a denial of service attack.

To avoid this DoS vector [depositTransaction \(opens in a new tab\)](#), and the functions that call it, require a gas limit parameter [This gas limit is encoded into the \[TransactionDeposited event \(opens in a new tab\)\]](#), and used as the gas limit for the user deposit transaction on L2.

This L2 gas is paid for by burning L1 gas [here \(opens in a new tab\)](#).

## Replaying messages

Deposits transactions can fail due to several reasons:

- Not enough gas provided.
- The state on L2 does not allow the transaction to be successful.

It is possible to replay a failed deposit, possibly with more gas.

## Replays in action

To see how replays work, you can use [this contract on OP Sepolia](#)(opens in a new tab).

[illegible]

```

96. input
97. `
98. CallstartChanges()
99. to allow changes using this Foundry command:
100. cast
101. send
102. --private-key
103. PRIV_KEY GREETER
104. "startChanges()"
105. ⚠
106. Don't do this prematurely
107. If you callstartChanges()
108. too early, it will happen when the message is relayed to L2, and then the initial deposit will be successful and there will be no need to replay it.
109. Verify thatgetStatus()
110. returns true, meaning changes are not allowed, and see the value ofgreet()
111. .
112. Foundry returns true as one.
113. cast
114. call
115. GREETER
116. "greet()"
117. |
118. cast
119. --to-ascii
120. ;
121. cast
122. call
123. GREETER
124. "getStatus()"
125. Now send the replay transaction.
126. cast
127. send
128. --private-key
129. PRIV_KEY
130. --gas-limit
131. 10000000
132. L2XDM_ADDRESS REPLAY_DATA
133. Why do we need to specify the gas limit?
134. The gas estimation mechanism tries to find the minimum gas limit at which the transaction would be successful.
135. However,L2CrossDomainMessenger
136. does not revert when a replay fails due to low gas limit, it just emits a failure message.
137. The gas estimation mechanism considers that a success.
138. To get a gas estimate, you can use this command:
139. cast
140. estimate
141. --from
142. 0x0000000000000000000000000000000000000000000000000000000000000001
143. L2XDM_ADDRESS REPLAY_DATA
144. That address is a special case in which the contract does revert.
145. Verify the greeting has changed:
146. cast
147. call
148. GREETER
149. "greet()"
150. |
151. cast
152. --to-ascii
153. ;
154. cast
155. call
156. GREETER
157. "getStatus()"

```

## Debugging

To debug deposit transactions you can ask the L2 cross domain messenger for the state of the transaction.

1. Look on Etherscan to see theFailedRelayedMessage
2. event.
3. SetMSG\_HASH
4. to that value.
5. To check if the message is listed as failed, run this:
6. cast
7. call
8. L2XDM\_ADDRESS
9. "failedMessages(bytes32)"
10. MSG\_HASH
11. To check if it is listed as successful, run this:
12. cast
13. call
14. L2XDM\_ADDRESS
15. "successfulMessages(bytes32)"
16. MSG\_HASH

[Smart Contracts Transaction Flow](#)