**Solidity - Frequently used Program Applications**

**Table of Contents**

* **Ether Wallet**
* **Multi Sig Wallet**
* **Merkle Tree**
* **Iterable Mapping**
* **ERC20**
* **ERC721**
* **Uni-directional Payment Channel**
* **Bi-directional Payment Channel**
* **NFT Auction**
* **Crowd Fund**
* **Time Lock**

**Ether Wallet**

Ethereum wallets are applications that let you interact with your Ethereum account. Think of it like an internet banking app – without the bank. Your wallet lets you read your balance, send transactions and connect to applications.

An example of a basic Ether wallet:

* Anyone can send ETH.
* Only the owner can withdraw.

Solidity code to implement a basic Ether wallet:

| // SPDX-License-Identifier: MIT pragma solidity ^0.8.0;  contract EtherWallet {  address payable public owner;   constructor() {  owner = payable(msg.sender);  }   receive() external payable {}   function withdraw(uint \_amount) external {  require(msg.sender == owner, "caller is not owner");  payable(msg.sender).transfer(\_amount);  }   function getBalance() external view returns (uint) {  return address(this).balance;  } } |
| --- |

**Multisig Wallet**

Multisig is an abbreviation for multi-signature, which is a sort of digital signature that allows two or more users to sign documents together as a group. Therefore, a multi-signature is created by combining numerous distinct signatures into a single document. Even while multisig technology has been around for a while in the realm of cryptocurrencies, its underlying idea dates back well before the invention of the Bitcoin currency.

As for cryptocurrencies, the concept was first applied to Bitcoin addresses in 2012, which finally led to the establishment of multisig wallets the following year. Multisig addresses can be utilized in a variety of situations, although the vast majority of them are related to security issues. In this section, we will examine their application in cryptocurrency wallets.

What is the procedure?

As a simple illustration, consider a safe deposit box with two locks and two keys. Alice is in possession of one key, and Bob is in possession of the other. The only way they can open the box is if they both provide their keys at the same moment, which means that one cannot access the box without the approval of the other.

Essentially, funds saved on a multi-signature address can only be retrieved by using two or more signatures, unless the address is changed. As a result, the usage of a multisig wallet enables users to add an additional layer of security to their funds, increasing their overall security. However, before proceeding, it is necessary to grasp the fundamentals of a normal Bitcoin address, which relies on a single key rather than many keys (single-key address).

Single-key versus multisignature

Standard, single-key addresses for Bitcoin are used to store the digital currency. This means that anyone who has access to that address's associated private key is able to access the money. In practice, this implies that only one key is required to sign transactions, and anyone who has the private key has the ability to transfer the money at their leisure, without the need for permission from anyone else.

While managing a single-key address is more convenient and faster than managing a multisig address, it comes with a number of drawbacks, particularly in terms of security. It is because of this single point of failure that the funds are safeguarded, which is why fraudsters are continuously devising new phishing schemes to attempt to steal the bitcoins and other cryptocurrencies that are being used.

Furthermore, single-key addresses are not the greatest option for enterprises who are involved in cryptocurrency transactions and exchanges. Consider the scenario in which the funds of a large corporation are kept on a standard address with a single associated private key. Essentially, this would mean that the private key would be handed to either a single individual or to a group of individuals at the same time - neither of which is the most secure option.

Multisig wallets have the ability to provide a solution to both of these issues. It is not possible to move cash stored on a multisig address without providing multiple signatures, which is in contrast to single-key (which are generated through the use of different private keys).

It is possible that a multisig address will require a different combination of keys depending on how it is configured: One of the most prevalent types of 2-of-3 addresses is one in which just two signatures are required to access the funds of a three-signature address. However, there are numerous different permutations, such as 2-of-2, 3-of-3, 3-of-4, and so on, that can be played.

There are a plethora of potential uses for this technological advancement. Here are a few of the most prevalent scenarios in which multi-signature bitcoin wallets are employed.

Increasing the level of security

By utilizing a multisig wallet, users can avoid the difficulties that can arise as a result of the loss or theft of a private key. Because of this, even if one of the keys is compromised, the money is still protected.

Consider the following scenario: Alice establishes a 2-of-3 multisig address and then saves each private key in a separate location or device (e.g. mobile phone, laptop, and tablet). Even if her mobile device is taken, the thief will not be able to access her cash if he or she just has one of the three keys in their possession. Additionally, phishing assaults and malware infections are less likely to succeed because the

hacker would most likely only have access to a single device and its associated encryption key.

Even if Alice loses one of her private keys as a result of a malicious attack, she will still be able to access her funds using the other two keys.

Two-factor authentication (also known as two-factor authentication)

Alice is able to construct a two-factor authentication mechanism for accessing her funds by establishing a multisig wallet with two keys that requires two keys. In this case, she may have one private key kept on her laptop and another stored on her mobile device (or even on a piece of paper). Only someone who has access to both keys would be able to complete a transaction as a result of this arrangement.

Always keep in mind, however, that utilizing multisignature technology as a two-factor authentication method can be risky, particularly if the address is configured as a 2-of-2 multisignature address. If you misplace one of the keys, you will not be able to access your funds until you replace it. As a result, employing a two-of-three setup or a third-party two-factor authentication solution that includes backup codes would be safer. Google Authenticator is highly recommended for crypto exchange trading accounts when it comes to ensuring the security of your accounts.

Transactions that are held in escrow

In the instance of an escrow transaction between two parties (Alice and Bob), the creation of a 2-of-3 multisig wallet can allow for the inclusion of a third person (Charlie) as a mutually trusted arbitrator in the event that something goes wrong.

In such a scenario, Alice would first deposit the funds, which would then be placed in a trust account (neither user being able to access them on their own). Then, if Bob provides the products or services in accordance with the agreement, they can both use their keys to sign and close the transaction.

In the event of a disagreement, Charlie, the arbiter, would only be required to intervene, at which time he may use his key to generate a signature that would be sent to either Alice or Bob, depending on Charlie's decision.

The process of making a decision

A multisig wallet may be used by a company's board of directors to control access to the company's cash. For example, by establishing a 4-of-6 wallet in which each board member has one key, it is impossible for any individual board member to misuse the cash. As a result, only choices that have been unanimously approved by the majority can be implemented.

Disadvantages of Multisig Wallets

Despite the fact that multisig wallets are a suitable solution for a variety of problems, it is crucial to remember that there are some dangers and limits associated with using them. A multisig address requires some technical knowledge, especially if you do not want to rely on third-party service providers for the setup.

Furthermore, because blockchain technology and multisignature addresses are both relatively new concepts, it may be difficult to seek legal remedy if something goes wrong with the system. There is no legal stewardship of funds deposited into a shared wallet with many keyholders because there is no legal stewardship.

The code below implements multisig wallet in solidity with the following specifications:

The wallet owners can

* submit a transaction
* approve and revoke approval of pending transactions
* anyone can execute a transaction after enough owners have approved it.

| // SPDX-License-Identifier: MIT pragma solidity ^0.8.0;  contract MultiSigWallet {  event Deposit(address indexed sender, uint amount, uint balance);  event SubmitTransaction(  address indexed owner,  uint indexed txIndex,  address indexed to,  uint value,  bytes data  );  event ConfirmTransaction(address indexed owner, uint indexed txIndex);  event RevokeConfirmation(address indexed owner, uint indexed txIndex);  event ExecuteTransaction(address indexed owner, uint indexed txIndex);   address[] public owners;  mapping(address => bool) public isOwner;  uint public numConfirmationsRequired;   struct Transaction {  address to;  uint value;  bytes data;  bool executed;  uint numConfirmations;  }   // mapping from tx index => owner => bool  mapping(uint => mapping(address => bool)) public isConfirmed;   Transaction[] public transactions;   modifier onlyOwner() {  require(isOwner[msg.sender], "not owner");  \_;  }   modifier txExists(uint \_txIndex) {  require(\_txIndex < transactions.length, "tx does not exist");  \_;  }   modifier notExecuted(uint \_txIndex) {  require(!transactions[\_txIndex].executed, "tx already executed");  \_;  }   modifier notConfirmed(uint \_txIndex) {  require(!isConfirmed[\_txIndex][msg.sender], "tx already confirmed");  \_;  }   constructor(address[] memory \_owners, uint \_numConfirmationsRequired) {  require(\_owners.length > 0, "owners required");  require(  \_numConfirmationsRequired > 0 &&  \_numConfirmationsRequired <= \_owners.length,  "invalid number of required confirmations"  );   for (uint i = 0; i < \_owners.length; i++) {  address owner = \_owners[i];   require(owner != address(0), "invalid owner");  require(!isOwner[owner], "owner not unique");   isOwner[owner] = true;  owners.push(owner);  }   numConfirmationsRequired = \_numConfirmationsRequired;  }   receive() external payable {  emit Deposit(msg.sender, msg.value, address(this).balance);  }   function submitTransaction(  address \_to,  uint \_value,  bytes memory \_data  ) public onlyOwner {  uint txIndex = transactions.length;   transactions.push(  Transaction({  to: \_to,  value: \_value,  data: \_data,  executed: false,  numConfirmations: 0  })  );   emit SubmitTransaction(msg.sender, txIndex, \_to, \_value, \_data);  }   function confirmTransaction(uint \_txIndex)  public  onlyOwner  txExists(\_txIndex)  notExecuted(\_txIndex)  notConfirmed(\_txIndex)  {  Transaction storage transaction = transactions[\_txIndex];  transaction.numConfirmations += 1;  isConfirmed[\_txIndex][msg.sender] = true;   emit ConfirmTransaction(msg.sender, \_txIndex);  }   function executeTransaction(uint \_txIndex)  public  onlyOwner  txExists(\_txIndex)  notExecuted(\_txIndex)  {  Transaction storage transaction = transactions[\_txIndex];   require(  transaction.numConfirmations >= numConfirmationsRequired,  "cannot execute tx"  );   transaction.executed = true;   (bool success, ) = transaction.to.call{value: transaction.value}(  transaction.data  );  require(success, "tx failed");   emit ExecuteTransaction(msg.sender, \_txIndex);  }   function revokeConfirmation(uint \_txIndex)  public  onlyOwner  txExists(\_txIndex)  notExecuted(\_txIndex)  {  Transaction storage transaction = transactions[\_txIndex];   require(isConfirmed[\_txIndex][msg.sender], "tx not confirmed");   transaction.numConfirmations -= 1;  isConfirmed[\_txIndex][msg.sender] = false;   emit RevokeConfirmation(msg.sender, \_txIndex);  }   function getOwners() public view returns (address[] memory) {  return owners;  }   function getTransactionCount() public view returns (uint) {  return transactions.length;  }   function getTransaction(uint \_txIndex)  public  view  returns (  address to,  uint value,  bytes memory data,  bool executed,  uint numConfirmations  )  {  Transaction storage transaction = transactions[\_txIndex];   return (  transaction.to,  transaction.value,  transaction.data,  transaction.executed,  transaction.numConfirmations  );  } } |
| --- |

Here is a contract to test sending transactions from the multi-sig wallet:

| // SPDX-License-Identifier: MIT pragma solidity ^0.8.0;  contract TestContract {  uint public i;   function callMe(uint j) public {  i += j;  }   function getData() public pure returns (bytes memory) {  return abi.encodeWithSignature("callMe(uint256)", 123);  } } |
| --- |

**Merkle Tree**

A blockchain is made up of a number of blocks that are connected to one another by links (hence the name blockchain). Using a hash tree, also known as the Merkle tree, blockchain data can be encoded in an efficient and secure manner. When used in conjunction with a peer-to-peer blockchain network, it allows for the rapid verification and movement of enormous volumes of data from one computer node to another on the network.

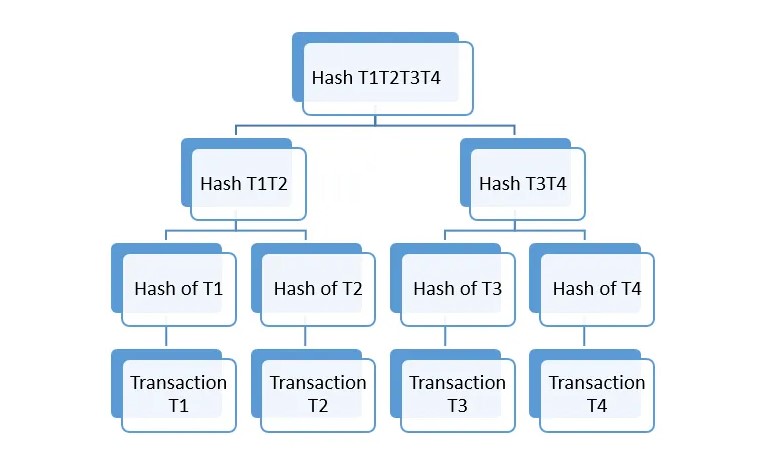
Almost every transaction that takes place on the blockchain network is associated with a hash value. On the block, these hashes are not recorded in a sequential sequence, but rather in the form of a tree-like structure, with each hash linked to its parent through a parent-child tree-like relationship between the two hashes.

Because there are numerous transactions stored on a single block, all of the transaction hashes contained within the block are also hashed, resulting in a Merkle root being generated.

Consider the case of a transaction block with seven transactions. There will be four transaction hashes at the lowest level of the hierarchy (referred to as the leaf level). There will be two transaction hashes at the level one above the leaf level, each of which will connect to two hashes that are below them at the leaf level, at the level one above the leaf level. At the very top (level two), there will be the last transaction hash, which will be referred to as the root, and it will connect to the two hashes below it by a network of connections (at level one).

To put it another way, you end up with an upside-down binary tree, with each node of the tree connecting to only two nodes below it (hence the name "binary tree"). A single root hash is located at the top of the structure, which connects to two hashes at level one, each of which connects to the two hashes at level three (leaf-level), and the structure continues indefinitely depending on the number of transaction hashes present.

Merkle tree example:



The hashing process begins with the nodes at the lowest level (leaf level), and all four hashes are included in the hash of nodes that are related to it at the first level. Similarly, hashing begins at level one, which leads to hashes of hashes reaching higher levels, until it reaches a single top root hash, which is the final result.

This root hash is referred to as the Merkle root, and it holds all of the information about every single transaction hash that appears on the block due to the tree-like connectivity of hashes that exists between them. It provides a single-point hash value that allows you to validate anything that is contained within that block.

Consider the following scenario: If someone has to validate a transaction that claims to have originated in block #137, they only need to look at the block's Merkle tree, and they don't have to worry about confirming anything in any other blocks on the blockchain, such as blocks #136 or #138.

Enter the Merkle root, which expedites the verification process even further. The fact that it contains all of the information about the entire tree means that all that is required is to check that transaction hash, its sister node (if it exists), and then move upward until it reaches the top of the tree

Essentially, the Merkle tree and Merkle root mechanisms greatly minimize the amount of hashing that must be done, allowing for speedier verification and transaction processing.

The following program implements merkle tree in Solidity:

| // SPDX-License-Identifier: MIT pragma solidity ^0.8.0;  contract MerkleProof {  function verify(  bytes32[] memory proof,  bytes32 root,  bytes32 leaf,  uint index  ) public pure returns (bool) {  bytes32 hash = leaf;   for (uint i = 0; i < proof.length; i++) {  bytes32 proofElement = proof[i];   if (index % 2 == 0) {  hash = keccak256(abi.encodePacked(hash, proofElement));  } else {  hash = keccak256(abi.encodePacked(proofElement, hash));  }   index = index / 2;  }   return hash == root;  } }  contract TestMerkleProof is MerkleProof {  bytes32[] public hashes;   constructor() {  string[4] memory transactions = [  "alice -> bob",  "bob -> dave",  "carol -> alice",  "dave -> bob"  ];   for (uint i = 0; i < transactions.length; i++) {  hashes.push(keccak256(abi.encodePacked(transactions[i])));  }   uint n = transactions.length;  uint offset = 0;   while (n > 0) {  for (uint i = 0; i < n - 1; i += 2) {  hashes.push(  keccak256(  abi.encodePacked(hashes[offset + i], hashes[offset + i + 1])  )  );  }  offset += n;  n = n / 2;  }  }   function getRoot() public view returns (bytes32) {  return hashes[hashes.length - 1];  }   /\* verify  3rd leaf  0x1bbd78ae6188015c4a6772eb1526292b5985fc3272ead4c65002240fb9ae5d13   root  0x074b43252ffb4a469154df5fb7fe4ecce30953ba8b7095fe1e006185f017ad10   index  2   proof  0x948f90037b4ea787c14540d9feb1034d4a5bc251b9b5f8e57d81e4b470027af8  0x63ac1b92046d474f84be3aa0ee04ffe5600862228c81803cce07ac40484aee43  \*/ } |
| --- |

**Iterable Mapping**

You cannot directly iterate through a mapping in Solidity. So here is an example of how to create an iterable mapping:

| // SPDX-License-Identifier: MIT pragma solidity ^0.8.0;  library IterableMapping {  // Iterable mapping from address to uint;  struct Map {  address[] keys;  mapping(address => uint) values;  mapping(address => uint) indexOf;  mapping(address => bool) inserted;  }   function get(Map storage map, address key) public view returns (uint) {  return map.values[key];  }   function getKeyAtIndex(Map storage map, uint index) public view returns (address) {  return map.keys[index];  }   function size(Map storage map) public view returns (uint) {  return map.keys.length;  }   function set(  Map storage map,  address key,  uint val  ) public {  if (map.inserted[key]) {  map.values[key] = val;  } else {  map.inserted[key] = true;  map.values[key] = val;  map.indexOf[key] = map.keys.length;  map.keys.push(key);  }  }   function remove(Map storage map, address key) public {  if (!map.inserted[key]) {  return;  }   delete map.inserted[key];  delete map.values[key];   uint index = map.indexOf[key];  uint lastIndex = map.keys.length - 1;  address lastKey = map.keys[lastIndex];   map.indexOf[lastKey] = index;  delete map.indexOf[key];   map.keys[index] = lastKey;  map.keys.pop();  } }  contract TestIterableMap {  using IterableMapping for IterableMapping.Map;   IterableMapping.Map private map;   function testIterableMap() public {  map.set(address(0), 0);  map.set(address(1), 100);  map.set(address(2), 200); // insert  map.set(address(2), 200); // update  map.set(address(3), 300);   for (uint i = 0; i < map.size(); i++) {  address key = map.getKeyAtIndex(i);   assert(map.get(key) == i \* 100);  }   map.remove(address(1));   // keys = [address(0), address(3), address(2)]  assert(map.size() == 3);  assert(map.getKeyAtIndex(0) == address(0));  assert(map.getKeyAtIndex(1) == address(3));  assert(map.getKeyAtIndex(2) == address(2));  } } |
| --- |

**ERC20 Tokens**

Ethereum, a famous cryptocurrency and blockchain system, is built on the use of tokens, which may be purchased, sold, or exchanged on the Ethereum exchange. Ethereum was first introduced in 2015, and since then it has grown to become one of the most important factors in the rise of cryptocurrency popularity. When used in the Ethereum system, tokens can represent a wide variety of digital assets, including vouchers, IOUs, and even physical, actual goods in the real world. Ethereum tokens, in their most basic form, are smart contracts that make use of the Ethereum network.

The ERC-20 token is considered to be one of the most important Ethereum tokens. This standard has emerged as the technical standard for token implementation on the Ethereum blockchain; it is used for all smart contracts on the blockchain, and it specifies a set of criteria that all Ethereum-based tokens must follow in order to be considered valid.

ERC-20 is similar to bitcoin, Litecoin, and any other cryptocurrency in that it is a blockchain-based asset that has value and can be transmitted and received; however, ERC-20 tokens are not a cryptocurrency in the traditional sense. The key distinction between ERC-20 tokens and other cryptographic tokens is that they are released on the Ethereum network rather than on their own blockchain.

ERC-20 establishes a standardized set of rules.

As of March 24, 2022, there are around 508,074 ERC-20-compatible tokens on Ethereum's main network, according to CoinMarketCap.

2 The ERC-20 standard is critical because it establishes a set of rules that all Ethereum tokens must follow in order to function properly. Some of these regulations govern how tokens can be moved, how transactions are approved, how users can access information about a token, and the overall number of tokens available for circulation.

As a result, this special token standard provides developers of all kinds with the ability to precisely predict how new tokens will function inside the broader Ethereum system. For developers, this simplifies the process at hand; they can proceed with their work confident that each and every new project will not need to be rewritten every time a new token is published, as long as the token complies with the rules. This compliance is also required since it assures that the numerous different tokens created on Ethereum are compatible with one another.

Unfortunately, the vast majority of token developers have failed to comply with ERC-20 guidelines, resulting in the fact that the vast majority of tokens issued through Ethereum initial coin offers (ICOs) are not ERC-20 compliant.

A large number of well-known digital currencies, such as Maker (MKR), Basic Attention Token (BAT), Augur (REP), and OmiseGO (OMG), are built on the ERC-20 standard (OMG). If you intend to purchase any digital currency that has been released as an ERC-20 token, you must also have a wallet that is compatible with these tokens in order to complete the transaction. The fact that ERC-20 tokens are so popular means that there are many different wallet alternatives to choose from.

**The following codes implement ERC20 token in Solidity**

**ERC20 Interface:**

| **// SPDX-License-Identifier: MIT pragma solidity ^0.8.0;  // https://github.com/OpenZeppelin/openzeppelin-contracts/blob/v3.0.0/contracts/token/ERC20/IERC20.sol interface IERC20 {  function totalSupply() external view returns (uint);   function balanceOf(address account) external view returns (uint);   function transfer(address recipient, uint amount) external returns (bool);   function allowance(address owner, address spender) external view returns (uint);   function approve(address spender, uint amount) external returns (bool);   function transferFrom(  address sender,  address recipient,  uint amount  ) external returns (bool);   event Transfer(address indexed from, address indexed to, uint value);  event Approval(address indexed owner, address indexed spender, uint value); }** |
| --- |

**Example of ERC20 token contract**

| **// SPDX-License-Identifier: MIT pragma solidity ^0.8.0;  import "./IERC20.sol";  contract ERC20 is IERC20 {  uint public totalSupply;  mapping(address => uint) public balanceOf;  mapping(address => mapping(address => uint)) public allowance;  string public name = "Solidity by Example";  string public symbol = "SOLBYEX";  uint8 public decimals = 18;   function transfer(address recipient, uint amount) external returns (bool) {  balanceOf[msg.sender] -= amount;  balanceOf[recipient] += amount;  emit Transfer(msg.sender, recipient, amount);  return true;  }   function approve(address spender, uint amount) external returns (bool) {  allowance[msg.sender][spender] = amount;  emit Approval(msg.sender, spender, amount);  return true;  }   function transferFrom(  address sender,  address recipient,  uint amount  ) external returns (bool) {  allowance[sender][msg.sender] -= amount;  balanceOf[sender] -= amount;  balanceOf[recipient] += amount;  emit Transfer(sender, recipient, amount);  return true;  }   function mint(uint amount) external {  balanceOf[msg.sender] += amount;  totalSupply += amount;  emit Transfer(address(0), msg.sender, amount);  }   function burn(uint amount) external {  balanceOf[msg.sender] -= amount;  totalSupply -= amount;  emit Transfer(msg.sender, address(0), amount);  } }** |
| --- |

**Creating your own ERC20 token contract using openzeppelin:**

| // SPDX-License-Identifier: MIT pragma solidity ^0.8.0;  import "https://github.com/OpenZeppelin/openzeppelin-contracts/blob/v4.0.0/contracts/token/ERC20/ERC20.sol";  contract MyToken is ERC20 {  constructor(string memory name, string memory symbol) ERC20(name, symbol) {  // Mint 100 tokens to msg.sender  // Similar to how  // 1 dollar = 100 cents  // 1 token = 1 \* (10 \*\* decimals)  \_mint(msg.sender, 100 \* 10\*\*uint(decimals()));  } } |
| --- |

**Creating a contract to swap ERC20 tokens**

**Here is an example contract, TokenSwap, that can be used to trade one ERC20 token for another in the Ethereum blockchain.**

**This contract will trade tokens by using the swap function,which will transfer the token's value from the sender to the recipient.**

**In order for transferFrom to be successful, the sender must have more tokens in their balance than the maximum number of tokens allowed. TokenSwap can withdraw a quantity of tokens by calling approve first, then calling transfer after TokenSwap has called transfer.**

| **// SPDX-License-Identifier: MIT pragma solidity ^0.8.0;  import "https://github.com/OpenZeppelin/openzeppelin-contracts/blob/v4.0.0/contracts/token/ERC20/IERC20.sol";  /\* How to swap tokens  1. Alice has 100 tokens from AliceCoin, which is a ERC20 token. 2. Bob has 100 tokens from BobCoin, which is also a ERC20 token. 3. Alice and Bob wants to trade 10 AliceCoin for 20 BobCoin. 4. Alice or Bob deploys TokenSwap 5. Alice approves TokenSwap to withdraw 10 tokens from AliceCoin 6. Bob approves TokenSwap to withdraw 20 tokens from BobCoin 7. Alice or Bob calls TokenSwap.swap() 8. Alice and Bob traded tokens successfully. \*/  contract TokenSwap {  IERC20 public token1;  address public owner1;  uint public amount1;  IERC20 public token2;  address public owner2;  uint public amount2;   constructor(  address \_token1,  address \_owner1,  uint \_amount1,  address \_token2,  address \_owner2,  uint \_amount2  ) {  token1 = IERC20(\_token1);  owner1 = \_owner1;  amount1 = \_amount1;  token2 = IERC20(\_token2);  owner2 = \_owner2;  amount2 = \_amount2;  }   function swap() public {  require(msg.sender == owner1 || msg.sender == owner2, "Not authorized");  require(  token1.allowance(owner1, address(this)) >= amount1,  "Token 1 allowance too low"  );  require(  token2.allowance(owner2, address(this)) >= amount2,  "Token 2 allowance too low"  );   \_safeTransferFrom(token1, owner1, owner2, amount1);  \_safeTransferFrom(token2, owner2, owner1, amount2);  }   function \_safeTransferFrom(  IERC20 token,  address sender,  address recipient,  uint amount  ) private {  bool sent = token.transferFrom(sender, recipient, amount);  require(sent, "Token transfer failed");  } }** |
| --- |

**To implement the token swap, execute the following function call:**

| **transferFrom(address sender, address recipient, uint256 amount)** |
| --- |

**ERC721- The Non Fungible token Standard**

What is a Non-Fungible Token, and how does it work?

A Non-Fungible Token (NFT) is a unique identifier that can be used to identify anything or someone. It is ideal for usage on platforms that provide collectible objects, access keys, lottery tickets, numbered seats for concerts and sporting events, among other things.. This unique sort of Token has incredible potential, and as a result, it demands a legitimate Standard. The ERC-721 was created to address this issue.

What exactly is ERC-721?

To put it another way, the ERC-721 establishes a standard for NFT, which means that this sort of Token is distinct and can have a different value than another Token from the same Smart Contract, either owing to its age, rarity, or even something else, such as its visual appearance. What's that, a visual?

Yes! Because all NFTs have an uint256 variable named tokenId, the pair contract address, uint256 tokenId, and uint256 tokenId must be globally unique for any ERC-721 Contract. However, a decentralized application can include a "converter" that takes the tokenId as input and returns an image of anything interesting, such as zombies, weapons, abilities or adorable kittens.

The following code implements ERC721 token standard in Solidity:

| // SPDX-License-Identifier: MIT pragma solidity ^0.8.0;  interface IERC165 {  function supportsInterface(bytes4 interfaceID) external view returns (bool); }  interface IERC721 is IERC165 {  function balanceOf(address owner) external view returns (uint balance);   function ownerOf(uint tokenId) external view returns (address owner);   function safeTransferFrom(  address from,  address to,  uint tokenId  ) external;   function safeTransferFrom(  address from,  address to,  uint tokenId,  bytes calldata data  ) external;   function transferFrom(  address from,  address to,  uint tokenId  ) external;   function approve(address to, uint tokenId) external;   function getApproved(uint tokenId) external view returns (address operator);   function setApprovalForAll(address operator, bool \_approved) external;   function isApprovedForAll(address owner, address operator)  external  view  returns (bool); }  interface IERC721Receiver {  function onERC721Received(  address operator,  address from,  uint tokenId,  bytes calldata data  ) external returns (bytes4); }  contract ERC721 is IERC721 {  using Address for address;   event Transfer(address indexed from, address indexed to, uint indexed tokenId);  event Approval(  address indexed owner,  address indexed approved,  uint indexed tokenId  );  event ApprovalForAll(  address indexed owner,  address indexed operator,  bool approved  );   // Mapping from token ID to owner address  mapping(uint => address) private \_owners;   // Mapping owner address to token count  mapping(address => uint) private \_balances;   // Mapping from token ID to approved address  mapping(uint => address) private \_tokenApprovals;   // Mapping from owner to operator approvals  mapping(address => mapping(address => bool)) private \_operatorApprovals;   function supportsInterface(bytes4 interfaceId)  external  pure  override  returns (bool)  {  return  interfaceId == type(IERC721).interfaceId ||  interfaceId == type(IERC165).interfaceId;  }   function balanceOf(address owner) external view override returns (uint) {  require(owner != address(0), "owner = zero address");  return \_balances[owner];  }   function ownerOf(uint tokenId) public view override returns (address owner) {  owner = \_owners[tokenId];  require(owner != address(0), "token doesn't exist");  }   function isApprovedForAll(address owner, address operator)  external  view  override  returns (bool)  {  return \_operatorApprovals[owner][operator];  }   function setApprovalForAll(address operator, bool approved) external override {  \_operatorApprovals[msg.sender][operator] = approved;  emit ApprovalForAll(msg.sender, operator, approved);  }   function getApproved(uint tokenId) external view override returns (address) {  require(\_owners[tokenId] != address(0), "token doesn't exist");  return \_tokenApprovals[tokenId];  }   function \_approve(  address owner,  address to,  uint tokenId  ) private {  \_tokenApprovals[tokenId] = to;  emit Approval(owner, to, tokenId);  }   function approve(address to, uint tokenId) external override {  address owner = \_owners[tokenId];  require(  msg.sender == owner || \_operatorApprovals[owner][msg.sender],  "not owner nor approved for all"  );  \_approve(owner, to, tokenId);  }   function \_isApprovedOrOwner(  address owner,  address spender,  uint tokenId  ) private view returns (bool) {  return (spender == owner ||  \_tokenApprovals[tokenId] == spender ||  \_operatorApprovals[owner][spender]);  }   function \_transfer(  address owner,  address from,  address to,  uint tokenId  ) private {  require(from == owner, "not owner");  require(to != address(0), "transfer to the zero address");   \_approve(owner, address(0), tokenId);   \_balances[from] -= 1;  \_balances[to] += 1;  \_owners[tokenId] = to;   emit Transfer(from, to, tokenId);  }   function transferFrom(  address from,  address to,  uint tokenId  ) external override {  address owner = ownerOf(tokenId);  require(  \_isApprovedOrOwner(owner, msg.sender, tokenId),  "not owner nor approved"  );  \_transfer(owner, from, to, tokenId);  }   function \_checkOnERC721Received(  address from,  address to,  uint tokenId,  bytes memory \_data  ) private returns (bool) {  if (to.isContract()) {  return  IERC721Receiver(to).onERC721Received(  msg.sender,  from,  tokenId,  \_data  ) == IERC721Receiver.onERC721Received.selector;  } else {  return true;  }  }   function \_safeTransfer(  address owner,  address from,  address to,  uint tokenId,  bytes memory \_data  ) private {  \_transfer(owner, from, to, tokenId);  require(\_checkOnERC721Received(from, to, tokenId, \_data), "not ERC721Receiver");  }   function safeTransferFrom(  address from,  address to,  uint tokenId,  bytes memory \_data  ) public override {  address owner = ownerOf(tokenId);  require(  \_isApprovedOrOwner(owner, msg.sender, tokenId),  "not owner nor approved"  );  \_safeTransfer(owner, from, to, tokenId, \_data);  }   function safeTransferFrom(  address from,  address to,  uint tokenId  ) external override {  safeTransferFrom(from, to, tokenId, "");  }   function mint(address to, uint tokenId) external {  require(to != address(0), "mint to zero address");  require(\_owners[tokenId] == address(0), "token already minted");   \_balances[to] += 1;  \_owners[tokenId] = to;   emit Transfer(address(0), to, tokenId);  }   function burn(uint tokenId) external {  address owner = ownerOf(tokenId);   \_approve(owner, address(0), tokenId);   \_balances[owner] -= 1;  delete \_owners[tokenId];   emit Transfer(owner, address(0), tokenId);  } }  library Address {  function isContract(address account) internal view returns (bool) {  uint size;  assembly {  size := extcodesize(account)  }  return size > 0;  } } |
| --- |

**Uni-Directional Payment Channel**

Payment channels enable parties to send and receive Ether off-chain on a regular basis.

The following is an example of how this contract is used:

* Alice activates the contract, which she funds with the Ether she has on hand.
* Alice authorizes a payment by signing a message (off chain) and sending the signature to Bob, who is the recipient of the money.
* Bob receives his payment by providing the smart contract with the signed message he created.
* If Bob does not claim his payment, Alice will receive her Ether back at the end of the contract's expiration period.
* The payment channel is referred to as a unidirectional payment channel since the payment can only flow in a single direction from Alice to Bob.

**The following code implements uni-directional payment channel in Solidity:**

| **// SPDX-License-Identifier: MIT pragma solidity ^0.8.0;  import "github.com/OpenZeppelin/openzeppelin-contracts/blob/release-v4.5/contracts/utils/cryptography/ECDSA.sol"; import "github.com/OpenZeppelin/openzeppelin-contracts/blob/release-v4.5/contracts/security/ReentrancyGuard.sol";  contract UniDirectionalPaymentChannel is ReentrancyGuard {  using ECDSA for bytes32;   address payable public sender;  address payable public receiver;   uint private constant DURATION = 7 \* 24 \* 60 \* 60;  uint public expiresAt;   constructor(address payable \_receiver) payable {  require(\_receiver != address(0), "receiver = zero address");  sender = payable(msg.sender);  receiver = \_receiver;  expiresAt = block.timestamp + DURATION;  }   function \_getHash(uint \_amount) private view returns (bytes32) {  // NOTE: sign with address of this contract to protect agains  // replay attack on other contracts  return keccak256(abi.encodePacked(address(this), \_amount));  }   function getHash(uint \_amount) external view returns (bytes32) {  return \_getHash(\_amount);  }   function \_getEthSignedHash(uint \_amount) private view returns (bytes32) {  return \_getHash(\_amount).toEthSignedMessageHash();  }   function getEthSignedHash(uint \_amount) external view returns (bytes32) {  return \_getEthSignedHash(\_amount);  }   function \_verify(uint \_amount, bytes memory \_sig) private view returns (bool) {  return \_getEthSignedHash(\_amount).recover(\_sig) == sender;  }   function verify(uint \_amount, bytes memory \_sig) external view returns (bool) {  return \_verify(\_amount, \_sig);  }   function close(uint \_amount, bytes memory \_sig) external nonReentrant {  require(msg.sender == receiver, "!receiver");  require(\_verify(\_amount, \_sig), "invalid sig");   (bool sent, ) = receiver.call{value: \_amount}("");  require(sent, "Failed to send Ether");  selfdestruct(sender);  }   function cancel() external {  require(msg.sender == sender, "!sender");  require(block.timestamp >= expiresAt, "!expired");  selfdestruct(sender);  } }** |
| --- |

**Bi-Directional Payment Channel**

Participants Alice and Bob can send Ether off the blockchain on a recurring basis thanks to bidirectional payment mechanisms.

Payments can be made in both directions; for example, Alice can pay Bob and Bob can pay Alice.

| // SPDX-License-Identifier: MIT pragma solidity ^0.8.0; pragma experimental ABIEncoderV2;  /\* Opening a channel 1. Alice and Bob fund a multi-sig wallet 2. Precompute payment channel address 3. Alice and Bob exchanges signatures of initial balances 4. Alice and Bob creates a transaction that can deploy a payment channel from  the multi-sig wallet  Update channel balances 1. Repeat steps 1 - 3 from opening a channel 2. From multi-sig wallet create a transaction that will  - delete the transaction that would have deployed the old payment channel  - and then create a transaction that can deploy a payment channel with the  new balances  Closing a channel when Alice and Bob agree on the final balance 1. From multi-sig wallet create a transaction that will  - send payments to Alice and Bob  - and then delete the transaction that would have created the payment channel  Closing a channel when Alice and Bob do not agree on the final balances 1. Deploy payment channel from multi-sig 2. call challengeExit() to start the process of closing a channel 3. Alice and Bob can withdraw funds once the channel is expired \*/  import "github.com/OpenZeppelin/openzeppelin-contracts/blob/release-v4.5/contracts/utils/math/SafeMath.sol"; import "github.com/OpenZeppelin/openzeppelin-contracts/blob/release-v4.5/contracts/utils/cryptography/ECDSA.sol";  contract BiDirectionalPaymentChannel {  using SafeMath for uint;  using ECDSA for bytes32;   event ChallengeExit(address indexed sender, uint nonce);  event Withdraw(address indexed to, uint amount);   address payable[2] public users;  mapping(address => bool) public isUser;   mapping(address => uint) public balances;   uint public challengePeriod;  uint public expiresAt;  uint public nonce;   modifier checkBalances(uint[2] memory \_balances) {  require(  address(this).balance >= \_balances[0].add(\_balances[1]),  "balance of contract must be >= to the total balance of users"  );  \_;  }   // NOTE: deposit from multi-sig wallet  constructor(  address payable[2] memory \_users,  uint[2] memory \_balances,  uint \_expiresAt,  uint \_challengePeriod  ) payable checkBalances(\_balances) {  require(\_expiresAt > block.timestamp, "Expiration must be > now");  require(\_challengePeriod > 0, "Challenge period must be > 0");   for (uint i = 0; i < \_users.length; i++) {  address payable user = \_users[i];   require(!isUser[user], "user must be unique");  users[i] = user;  isUser[user] = true;   balances[user] = \_balances[i];  }   expiresAt = \_expiresAt;  challengePeriod = \_challengePeriod;  }   function verify(  bytes[2] memory \_signatures,  address \_contract,  address[2] memory \_signers,  uint[2] memory \_balances,  uint \_nonce  ) public pure returns (bool) {  for (uint i = 0; i < \_signatures.length; i++) {  /\*  NOTE: sign with address of this contract to protect  agains replay attack on other contracts  \*/  bool valid = \_signers[i] ==  keccak256(abi.encodePacked(\_contract, \_balances, \_nonce))  .toEthSignedMessageHash()  .recover(\_signatures[i]);   if (!valid) {  return false;  }  }   return true;  }   modifier checkSignatures(  bytes[2] memory \_signatures,  uint[2] memory \_balances,  uint \_nonce  ) {  // Note: copy storage array to memory  address[2] memory signers;  for (uint i = 0; i < users.length; i++) {  signers[i] = users[i];  }   require(  verify(\_signatures, address(this), signers, \_balances, \_nonce),  "Invalid signature"  );   \_;  }   modifier onlyUser() {  require(isUser[msg.sender], "Not user");  \_;  }   function challengeExit(  uint[2] memory \_balances,  uint \_nonce,  bytes[2] memory \_signatures  )  public  onlyUser  checkSignatures(\_signatures, \_balances, \_nonce)  checkBalances(\_balances)  {  require(block.timestamp < expiresAt, "Expired challenge period");  require(\_nonce > nonce, "Nonce must be greater than the current nonce");   for (uint i = 0; i < \_balances.length; i++) {  balances[users[i]] = \_balances[i];  }   nonce = \_nonce;  expiresAt = block.timestamp.add(challengePeriod);   emit ChallengeExit(msg.sender, nonce);  }   function withdraw() public onlyUser {  require(block.timestamp >= expiresAt, "Challenge period has not expired yet");   uint amount = balances[msg.sender];  balances[msg.sender] = 0;   (bool sent, ) = msg.sender.call{value: amount}("");  require(sent, "Failed to send Ether");   emit Withdraw(msg.sender, amount);  } } |
| --- |

NFT Auction

Creating a NFT auction contract

* This contract is implemented by the NFT seller.
* The bidding is open for seven days
* Those interested in bidding can do so by making an ETH deposit equal to or more than the highest bid.
* Anyone who has placed an offer can retract it if it is not the highest bid.

After the auction:

* Immediately following the public auction,the NFT is sold to the highest bidder.
* It's a win-win situation for both the buyer and the seller.

| // SPDX-License-Identifier: MIT pragma solidity ^0.8.0;  interface IERC721 {  function safeTransferFrom(  address from,  address to,  uint tokenId  ) external;   function transferFrom(  address,  address,  uint  ) external; }  contract EnglishAuction {  event Start();  event Bid(address indexed sender, uint amount);  event Withdraw(address indexed bidder, uint amount);  event End(address winner, uint amount);   IERC721 public nft;  uint public nftId;   address payable public seller;  uint public endAt;  bool public started;  bool public ended;   address public highestBidder;  uint public highestBid;  mapping(address => uint) public bids;   constructor(  address \_nft,  uint \_nftId,  uint \_startingBid  ) {  nft = IERC721(\_nft);  nftId = \_nftId;   seller = payable(msg.sender);  highestBid = \_startingBid;  }   function start() external {  require(!started, "started");  require(msg.sender == seller, "not seller");   nft.transferFrom(msg.sender, address(this), nftId);  started = true;  endAt = block.timestamp + 7 days;   emit Start();  }   function bid() external payable {  require(started, "not started");  require(block.timestamp < endAt, "ended");  require(msg.value > highestBid, "value < highest");   if (highestBidder != address(0)) {  bids[highestBidder] += highestBid;  }   highestBidder = msg.sender;  highestBid = msg.value;   emit Bid(msg.sender, msg.value);  }   function withdraw() external {  uint bal = bids[msg.sender];  bids[msg.sender] = 0;  payable(msg.sender).transfer(bal);   emit Withdraw(msg.sender, bal);  }   function end() external {  require(started, "not started");  require(block.timestamp >= endAt, "not ended");  require(!ended, "ended");   ended = true;  if (highestBidder != address(0)) {  nft.safeTransferFrom(address(this), highestBidder, nftId);  seller.transfer(highestBid);  } else {  nft.safeTransferFrom(address(this), seller, nftId);  }   emit End(highestBidder, highestBid);  } } |
| --- |

**Crowdfunding**

Crowd fund ERC20 token

1. User creates a campaign.
2. Users can pledge, transferring their token to a campaign.
3. After the campaign ends, the campaign creator can claim the funds if the total amount pledged is more than the campaign goal.
4. Otherwise, the campaign did not reach its goal, users can withdraw their pledge.

| // SPDX-License-Identifier: MIT pragma solidity ^0.8.0;  interface IERC20 {  function transfer(address, uint) external returns (bool);   function transferFrom(  address,  address,  uint  ) external returns (bool); }  contract CrowdFund {  event Launch(  uint id,  address indexed creator,  uint goal,  uint32 startAt,  uint32 endAt  );  event Cancel(uint id);  event Pledge(uint indexed id, address indexed caller, uint amount);  event Unpledge(uint indexed id, address indexed caller, uint amount);  event Claim(uint id);  event Refund(uint id, address indexed caller, uint amount);   struct Campaign {  // Creator of campaign  address creator;  // Amount of tokens to raise  uint goal;  // Total amount pledged  uint pledged;  // Timestamp of start of campaign  uint32 startAt;  // Timestamp of end of campaign  uint32 endAt;  // True if goal was reached and creator has claimed the tokens.  bool claimed;  }   IERC20 public immutable token;  // Total count of campaigns created.  // It is also used to generate id for new campaigns.  uint public count;  // Mapping from id to Campaign  mapping(uint => Campaign) public campaigns;  // Mapping from campaign id => pledger => amount pledged  mapping(uint => mapping(address => uint)) public pledgedAmount;   constructor(address \_token) {  token = IERC20(\_token);  }   function launch(  uint \_goal,  uint32 \_startAt,  uint32 \_endAt  ) external {  require(\_startAt >= block.timestamp, "start at < now");  require(\_endAt >= \_startAt, "end at < start at");  require(\_endAt <= block.timestamp + 90 days, "end at > max duration");   count += 1;  campaigns[count] = Campaign({  creator: msg.sender,  goal: \_goal,  pledged: 0,  startAt: \_startAt,  endAt: \_endAt,  claimed: false  });   emit Launch(count, msg.sender, \_goal, \_startAt, \_endAt);  }   function cancel(uint \_id) external {  Campaign memory campaign = campaigns[\_id];  require(campaign.creator == msg.sender, "not creator");  require(block.timestamp < campaign.startAt, "started");   delete campaigns[\_id];  emit Cancel(\_id);  }   function pledge(uint \_id, uint \_amount) external {  Campaign storage campaign = campaigns[\_id];  require(block.timestamp >= campaign.startAt, "not started");  require(block.timestamp <= campaign.endAt, "ended");   campaign.pledged += \_amount;  pledgedAmount[\_id][msg.sender] += \_amount;  token.transferFrom(msg.sender, address(this), \_amount);   emit Pledge(\_id, msg.sender, \_amount);  }   function unpledge(uint \_id, uint \_amount) external {  Campaign storage campaign = campaigns[\_id];  require(block.timestamp <= campaign.endAt, "ended");   campaign.pledged -= \_amount;  pledgedAmount[\_id][msg.sender] -= \_amount;  token.transfer(msg.sender, \_amount);   emit Unpledge(\_id, msg.sender, \_amount);  }   function claim(uint \_id) external {  Campaign storage campaign = campaigns[\_id];  require(campaign.creator == msg.sender, "not creator");  require(block.timestamp > campaign.endAt, "not ended");  require(campaign.pledged >= campaign.goal, "pledged < goal");  require(!campaign.claimed, "claimed");   campaign.claimed = true;  token.transfer(campaign.creator, campaign.pledged);   emit Claim(\_id);  }   function refund(uint \_id) external {  Campaign memory campaign = campaigns[\_id];  require(block.timestamp > campaign.endAt, "not ended");  require(campaign.pledged < campaign.goal, "pledged >= goal");   uint bal = pledgedAmount[\_id][msg.sender];  pledgedAmount[\_id][msg.sender] = 0;  token.transfer(msg.sender, bal);   emit Refund(\_id, msg.sender, bal);  } } |
| --- |

**Timelock Contract**

A TimeLock contract publishes a transaction that will take place at some point in the future. It is possible to complete the transaction once the shortest possible amount of time has elapsed.

DAOs frequently make use of TimeLocks.

The following code implements timelock in Solidity

| // SPDX-License-Identifier: MIT pragma solidity ^0.8.0;  contract TimeLock {  error NotOwnerError();  error AlreadyQueuedError(bytes32 txId);  error TimestampNotInRangeError(uint blockTimestamp, uint timestamp);  error NotQueuedError(bytes32 txId);  error TimestampNotPassedError(uint blockTimestmap, uint timestamp);  error TimestampExpiredError(uint blockTimestamp, uint expiresAt);  error TxFailedError();   event Queue(  bytes32 indexed txId,  address indexed target,  uint value,  string func,  bytes data,  uint timestamp  );  event Execute(  bytes32 indexed txId,  address indexed target,  uint value,  string func,  bytes data,  uint timestamp  );  event Cancel(bytes32 indexed txId);   uint public constant MIN\_DELAY = 10; // seconds  uint public constant MAX\_DELAY = 1000; // seconds  uint public constant GRACE\_PERIOD = 1000; // seconds   address public owner;  // tx id => queued  mapping(bytes32 => bool) public queued;   constructor() {  owner = msg.sender;  }   modifier onlyOwner() {  if (msg.sender != owner) {  revert NotOwnerError();  }  \_;  }   receive() external payable {}   function getTxId(  address \_target,  uint \_value,  string calldata \_func,  bytes calldata \_data,  uint \_timestamp  ) public pure returns (bytes32) {  return keccak256(abi.encode(\_target, \_value, \_func, \_data, \_timestamp));  }   /\*\*  \* @param \_target Address of contract or account to call  \* @param \_value Amount of ETH to send  \* @param \_func Function signature, for example "foo(address,uint256)"  \* @param \_data ABI encoded data send.  \* @param \_timestamp Timestamp after which the transaction can be executed.  \*/  function queue(  address \_target,  uint \_value,  string calldata \_func,  bytes calldata \_data,  uint \_timestamp  ) external onlyOwner returns (bytes32 txId) {  txId = getTxId(\_target, \_value, \_func, \_data, \_timestamp);  if (queued[txId]) {  revert AlreadyQueuedError(txId);  }  // ---|------------|---------------|-------  // block block + min block + max  if (  \_timestamp < block.timestamp + MIN\_DELAY ||  \_timestamp > block.timestamp + MAX\_DELAY  ) {  revert TimestampNotInRangeError(block.timestamp, \_timestamp);  }   queued[txId] = true;   emit Queue(txId, \_target, \_value, \_func, \_data, \_timestamp);  }   function execute(  address \_target,  uint \_value,  string calldata \_func,  bytes calldata \_data,  uint \_timestamp  ) external payable onlyOwner returns (bytes memory) {  bytes32 txId = getTxId(\_target, \_value, \_func, \_data, \_timestamp);  if (!queued[txId]) {  revert NotQueuedError(txId);  }  // ----|-------------------|-------  // timestamp timestamp + grace period  if (block.timestamp < \_timestamp) {  revert TimestampNotPassedError(block.timestamp, \_timestamp);  }  if (block.timestamp > \_timestamp + GRACE\_PERIOD) {  revert TimestampExpiredError(block.timestamp, \_timestamp + GRACE\_PERIOD);  }   queued[txId] = false;   // prepare data  bytes memory data;  if (bytes(\_func).length > 0) {  // data = func selector + \_data  data = abi.encodePacked(bytes4(keccak256(bytes(\_func))), \_data);  } else {  // call fallback with data  data = \_data;  }   // call target  (bool ok, bytes memory res) = \_target.call{value: \_value}(data);  if (!ok) {  revert TxFailedError();  }   emit Execute(txId, \_target, \_value, \_func, \_data, \_timestamp);   return res;  }   function cancel(bytes32 \_txId) external onlyOwner {  if (!queued[\_txId]) {  revert NotQueuedError(\_txId);  }   queued[\_txId] = false;   emit Cancel(\_txId);  } } |
| --- |