

Lingo.sol : ERC20 Token Contract

● Token Basic Settings

1. Maximum token supply: 1 Billion
2. Initial token supply: 1 Billion
3. Token symbol: LINGO
4. Token name: LINGO
5. Number of decimals allowable: 18

● Token Features

1. **Total Supply:** Get the total token supply
2. **BalanceOf:** Get the token balance of an address
3. **Transfer:** Ability to send tokens to another address
4. **Approve:** Give another address permission to spend a certain allowance of tokens
5. **Allowance:** Check the allowance that one address has for another
6. **TransferFrom:** Spend the allowance that one address has for another
7. **Ownership:** The ownership module to change the owner and get the owner ID
8. **Burning:** The ability of the owner to reduce some tokens from the total supply for managing their economic model.
9. **Transaction fee:** For any transaction of LINGO token, [0 - 5%] (of the Token) will be deducted from the transaction.
An admin should be able to modify the transferFee, the value should be enforced by smart contract to not go outside of the [0 - 5%].
The transferFees should be transferred to a single transferFeeTreasuryWallet address. This address should be able to be modified by an admin
10. **Whitelisting:** A function allowing for an admin to whitelist addresses that are not going to pay transferFees if money is transferred to them. There should be 2 types of whitelisting. InternalWhitelistedAddresses and externalWhitelistedAddresses.

If the sender or receiver belongs to internalWhitelistedAddresses no fee will be debited from the transaction.

If the receiver belongs to externalWhitelistedAddresses no fee will be debited from the transaction.

11. **Minting:** Ability to add tokens to the initial allocation but to an address chosen by the Admin.

Distribution.sol: Reward calculation and distribution Contract

This contract is to stake LINGO tokens and manage reward distribution to the staked users. It keeps track of the credits and profits for each user and allows them to deposit, withdraw, and claim their earnings.

● Basic functionality

1. Users can deposit LINGO tokens to this contract
2. Every month the admin will provide an amount that will be distributed to these users who deposited their funds on contract.
3. Every month users can claim this fund from the contract.
4. Users can deposit more LINGO tokens at any time.
5. Users can withdraw LINGO tokens fully or partially at any time.
6. If the user has not claimed their reward for a month, it can be claimed along with next month's reward.
7. The admin can claim that if the user has not claimed their fund for a year.

● Important Functions and Variables

The reward will be calculated according to the credits that a user earned. Credit will be calculated by the following formula:

$$\text{Staked Amount} * \text{Time (In hours)}$$

The contract will keep track of each user's balance, the credits they have earned, and the last time they made a transaction and claimed their profits.

The explanation will be based on the javascript code.

1. State Variables

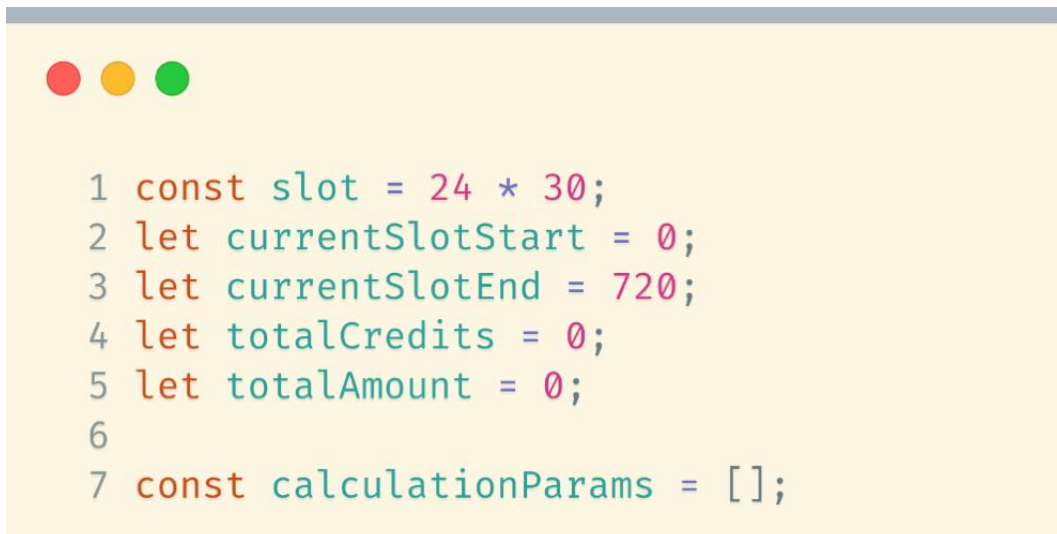
Each user information contains the following parameters:

```
1 const state = [  
2   {  
3     id: 'user1',  
4     balance: 0,  
5     forecastedCredits: 0,  
6     lastUpdatedTimestamp: 0,  
7     lastClaimedTimestamp: 0,  
8   },  
9   {  
10    id: 'user2',  
11    balance: 0,  
12    forecastedCredits: 0,  
13    lastUpdatedTimestamp: 0,  
14    lastClaimedTimestamp: 0,  
15  },  
16 ];
```

1. **Address:** Wallet address
2. **balance:** This is the amount of token that the user has in their account. It is a numeric value.

3. **lastUpdatedTimestamp**: This is the time when the user's balance was last updated. It is a numeric value representing the number of hours since a specific point in time or unix timestamp.
4. **lastClaimedTimestamp**: This is the time when the user last claimed their profits. It is a numeric value representing the number of hours since a specific point in time. It is initially set to 0.
5. **forecastedCredits**: The forecastedCredits variable is used to keep track of the credits that a user is expected to earn based on their current balance and the amount of time that has passed.

In addition to the user variables contract will also keep the following variables



```
1 const slot = 24 * 30;
2 let currentSlotStart = 0;
3 let currentSlotEnd = 720;
4 let totalCredits = 0;
5 let totalAmount = 0;
6
7 const calculationParams = [];
```

1. **Slot** is a constant variable that represents the number of hours in a slot, which is used as a unit of time measurement in this code. In this case, each slot is equivalent to 30 days or a month. So, the value of the slot is 24 (hours) multiplied by 30, which equals 720.
2. **currentSlotStart** is a variable that represents the start time of the current slot, measured in hours. It is initialized to 0 at the beginning of the program.
3. **currentSlotEnd** is a variable that represents the end time of the current slot, measured in hours. It is initialized to 720, which represents the end of the first slot, as each slot is 720 hours long.
4. **totalCredits** is a variable that keeps track of the total credits earned by all users in the current slot. It is initialized to 0 at the beginning of the program. It will be updated on deposit, withdraw and distribute function calls.
5. **totalAmount** is a variable that keeps track of the total amount deposited by all users in the current slot. It is initialized to 0 at the beginning of the program.
6. **calculationParams** is a variable that keeps track of the state of the contract each month. It will be updated on distribution function calls. It will be used to calculate the user rewards for past months if the user missed their monthly claims.

2. DepositFunction

1. Function to add LINGO tokens for staking.
2. Users can deposit any amount at any time.
3. The function updates the user's balance, recalculates their forecasted credits, and updates the global variables totalCredits and totalAmount. Finally, it updates the state array with the new user information.

```
1
2   totalCredits -= user.forecastedCredits;
3   user.forecastedCredits -= user.balance * (currentSlotEnd - timeInHours);
4
5   user.balance += amount;
6
7   user.forecastedCredits += user.balance * (currentSlotEnd - timeInHours);
8   totalCredits += user.forecastedCredits;
9   user.lastUpdatedTimestamp = timeInHours;
10
11  totalAmount += amount;
```

3. Withdraw Function

1. Users can withdraw any amount at any time.
2. A withdrawal fee will be associated with this withdrawal. (0 % - 5%). It is additional to transferFee.
3. The collected fee will be sent to treasuryWallet.
4. Treasury wallet and Fee percentage can be updated by Admin.
5. The withdraw function updates the state of the system by modifying the balance and forecastedCredits of the specified user and updating the totalCredits and totalAmount variables.

```
1   totalCredits -= user.forecastedCredits;
2   user.forecastedCredits -= user.balance * (currentSlotEnd - timeInHours);
3
4   user.balance -= amount;
5   user.forecastedCredits += user.balance * (currentSlotEnd - timeInHours);
6   user.lastUpdatedTimestamp = timeInHours;
7   totalCredits += user.forecastedCredits;
8
9   totalAmount -= amount;
10
11  state[userIndex] = user;
```

4. Distribute Function (admin only)

1. Admin will call the function at regular intervals (Once a month) with the amount collected for distribution.
2. Admin has to approve the contract before calling the function. Which will be validated.

3. Tokens released for claiming or unclaimed tokens will not be considered as a locked amount to calculate the next claim.
4. The distribute function is responsible for distributing the monthly profits generated from the users' balances to their forecasted credits. It takes an amount parameter as input, which represents the total monthly profit.
5. When the function is called, it first stores the current slot's start time, end time, monthly profit, and total credits in an array named calculationParams. Then, it updates the currentSlotStart and currentSlotEnd variables to represent the next time slot and calculates the total credits by multiplying the total amount by the slot duration.

```
1 calculationParams.push({
2     startTime: currentSlotStart,
3     endTime: currentSlotEnd,
4     monthlyProfit: amount,
5     totalCredits,
6 });
7 currentSlotStart = currentSlotEnd;
8 currentSlotEnd = currentSlotEnd + slot;
9 totalCredits = totalAmount * slot;
```

5. ClaimFunction

1. The claim function allows users to claim their share of the tokens that were distributed to the contract by the distribute function.
2. If the user has not claimed the allocation for a month, it will be added along with the next month's claim
3. Only transaction fees will be associated with the claim.
4. The function takes no parameters, as the contract knows which user is claiming their tokens based on the transaction sender's address.
5. The function first calculates the number of tokens that the user is entitled to based on their percentage of the total supply and the current balance of the contract.
6. If the user is entitled to tokens, the function transfers them from the contract's balance to the user's address using the transfer function.
7. If the user is not entitled to tokens, either because they have already claimed them or because there are no tokens left to claim, the function will revert the transaction and return an error message.
8. The function emits an event that includes the user's address, the number of tokens they claimed, and the total amount of tokens claimed so far.
9. Overall, the claim function ensures that users can receive their share of the tokens that were distributed to the contract, as long as they have not already claimed them. It also prevents users from claiming more tokens than they are entitled to or claiming tokens that have already been distributed.



```
1 const user = state[userIndex]
2 const { startTime, monthlyProfit, totalCredits } =
3   calculationParams[calculationParams.length - 1]
4
5 const claim = (user.forecastedCredits / totalCredits) * monthlyProfit
6
7 user.forecastedCredits = user.balance * slot
8 user.lastUpdatedTimestamp = timeInHours
9 user.lastClaimedTimestamp = timeInHours
10
11 state[userIndex] = user
12
```

6. PendingClaimFunction

1. The pending claim function is used when the user wants to claim credits for a time period that has already elapsed, but they did not claim credits at the start of the period. In this case, the function calculates the user's claim for all the elapsed periods since their last claimed timestamp and returns the total claim amount.
2. When called, the function first retrieves the calculation parameters for the periods that have elapsed since the user's last claimed timestamp, by filtering the calculationParams array. For each of these periods, the function calculates the user's claim as a proportion of their forecasted credits relative to the total credits for that period, multiplied by the monthly profit for that period. The function then adds up these claims to get the total claim amount for the user.
3. Finally, the function updates the user's forecastedCredits, lastUpdatedTimestamp, and lastClaimedTimestamp fields to reflect that they have now claimed their credits for the elapsed periods.

```

1  const pendingCalculationParams = calculationParams.filter(
2    (params) => params.startTime >= user.lastClaimedTimestamp
3  )
4
5  let totalClaim = 0
6
7  pendingCalculationParams.forEach(
8    ({ startTime, monthlyProfit, totalCredits }) => {
9      let forecastedCredits = 0
10
11      if (startTime === user.lastClaimedTimestamp) {
12        forecastedCredits = user.forecastedCredits
13      } else {
14        forecastedCredits = user.balance * slot
15      }
16
17      totalClaim += (forecastedCredits / totalCredits) * monthlyProfit
18
19      user.forecastedCredits = user.balance * slot
20      user.lastUpdatedTimestamp = timeInHours
21      user.lastClaimedTimestamp = timeInHours
22
23      state[userIndex] = user
24    }
25  )
26

```

7. Admin claim (admin only)

1. For the admin to claim the unclaimed funds.
2. If a user hasn't made any claim for 'X' months, the admin can claim the unclaimed rewards, not the locked fund.
3. This time can be updated by admin. It will be in the range of 1 - 10 years.
4. Admin can also disable this feature as required

There will be other functions and changes in the explained functions as well. But with these main things through we can achieve the requirements. For better understanding, we are adding some example calculations also below:

EXAMPLE

1. Initial State

```

const slot = 24 * 30;
let currentSlotStart = 0;
let currentSlotEnd = 720;
let totalCredits = 0;
let totalAmount = 0;

```

👉 - Initial state

(index)	id	balance	forecastedCredits	lastUpdatedTimestamp	lastClaimedTimestamp
0	'user1'	0	0	0	0
1	'user2'	0	0	0	0

2. User 1 deposits 1000 and User 2 deposits 500 at 0th hour. So the states and global variables will be like this:

(index)	id	balance	forecastedCredits	lastUpdatedTimestamp	lastClaimedTimestamp
0	'user1'	1000	720000	0	0
1	'user2'	500	360000	0	0

👉 - totalCredits: 1080000

👉 - totalAmount: 1500

Here forecastedCredits will be calculated as $1000 * \text{slot}$. (We will be the maximum credits per slot with current balance). Also adding these details we will update totalCredits and totalAmount.

3. Now after 10 days (240 hours) User 1 withdraws 500 from the funds. So state and variables will be like:

(index)	id	balance	forecastedCredits	lastUpdatedTimestamp	lastClaimedTimestamp
0	'user1'	500	480000	240	0
1	'user2'	500	360000	0	0

👉 - totalCredits: 840000

👉 - totalAmount: 1000

For user 1 it is like 1000 LINGO for 10 days and 500 LINGO for 20 days. So forecastedCredits, totalCredits and totalAmounts will be updated according to that.

4. Now say admin calls distribute at the 30th or 31st day with a total of 1000 LINGO tokens to distribute. So we store the slot details like this:

(index)	startTime	endTime	monthlyProfit	totalCredits
0	0	720	1000	840000

CASE 1 : Both Users Claim

5. Now both users can claim their tokens. Rewards will be calculated for user using the equation:

User 1 rewards = (User 1 forecastedCredits / totalCredits) * monthlyProfit

👉 - user1's credits : 480000
 👉 - user1's claim : 571.4285714285714

(index)	id	balance	forecastedCredits	lastUpdatedTimestamp	lastClaimedTimestamp
0	'user1'	500	360000	720	720
1	'user2'	500	360000	0	0

👉 - user2's credits : 360000
 👉 - user2's claim : 428.57142857142856

(index)	id	balance	forecastedCredits	lastUpdatedTimestamp	lastClaimedTimestamp
0	'user1'	500	360000	720	720
1	'user2'	500	360000	720	720

After the claim the forecastedCredits of them are reseted according to their current balance. Timestamps will also be updated. We will have the totalBalance at that time. From that we will calculate the totalCredits. (totalBalance * slot).

CASE 2 : Only User 1 Claims

6. ClaimdetailsofUser1andotherparametervaluesarelike

👉 - user1's credits : 480000
 👉 - user1's claim : 571.4285714285714

(index)	id	balance	forecastedCredits	lastUpdatedTimestamp	lastClaimedTimestamp
0	'user1'	500	360000	720	720
1	'user2'	500	360000	0	0

👉 - totalCredits: 720000
 👉 - totalAmount: 1000

7. Let's say they haven't made any transactions one 2nd month and user 2 try to claim on the second month. So first admin will add the profit,

👉 - distribute

(index)	startTime	endTime	monthlyProfit	totalCredits
0	0	720	1000	840000
1	720	1440	1000	720000

Then the claim details of both users will be like: For user 2 claim will be the sum of both months claims.

👉 - user1's credits : 360000

👉 - user1's claim : 500

(index)	id	balance	forecastedCredits	lastUpdatedTimestamp	lastClaimedTimestamp
0	'user1'	500	360000	1440	1440
1	'user2'	500	360000	0	0

👉 - user2's claim : 928.5714285714286

(index)	id	balance	forecastedCredits	lastUpdatedTimestamp	lastClaimedTimestamp
0	'user1'	500	360000	1440	1440
1	'user2'	500	360000	1440	1440

👉 - totalCredits: 720000

👉 - totalAmount: 1000