Specs of governance contracts

Contract `Timelock`

The contract inherits from the OpenZeppelin contract `TimelockController`.
The constructor of `Timelock` takes the inputs `minDelay, proposers, executors` and passes these parameters to the constructor of its parent contract `TimelockController`.

```
contract Timelock is TimelockController {
    constructor(uint256 minDelay, address[] memory proposers,
address[] memory executors)
        TimelockController(minDelay, proposers, executors)
    {}
}
```

Contract `GovernorOLAS`

The contract `GovernorOLAS` allows the deployment of voting protocols similar to Compound. This contract is derived from OpenZeppelin modular system of governance contracts (see Governance - OpenZeppelin for a detailed description). In particular, it imports the abstract contract `Governor` customized with the votes module `GovernorVotes` that, in this governance system, extracts voting weight from a `ERC20Votes` combined with `GovernorVotesQuorumFraction` to set the quorum as a fraction of the total token supply.
`GovernorOLA` inherits from the contract `GovernorTimelockControl` which binds the execution process to an instance of the `TimelockController`. The latter imposes a delay for governance decisions to be executed. Thus, the usual workflow is extended to require a **queue** step before **execution**. Moreover the timelock extension, in addition to the Governor itself, allows multiple proposers and executors.

`GovernorOLAS` inherited from `GovernorSetting` in order to manage some of the settings (voting delay, voting period duration, and proposal threshold) in a way that can be updated through a governance proposal, without requiring an upgrade.
Finally, the `GovernorOLAS` inherited from the contract `GovernorCompatibilityBravo` the GovernorBravo compatibility[1].

[1]Some examples of what Governor Bravo compatibility entails again. First of all the number of votes required in order for a voter to become a proposal, denoted by proposalThreshold(), is part of Governor Bravo's interface. Being bravo compatible the following

Dependencies

From OpenZeppelin, the contracts `Governor, GovernorSettings, GovernorCompatibilityBravo, GovernorVotes, GovernorVotesQuorumFraction, GovernorTimelockControl` are inherited (in this inheritance order[2]) by `GovernorOLAS`

```
contract GovernorOLAS is Governor, GovernorSettings,
GovernorCompatibilityBravo, GovernorVotes,
GovernorVotesQuorumFraction, GovernorTimelockControl
```

Constructor

The parent's constructor contracts are called in the inheritance order described above. The constructor of `GovernorOLAS` takes the inputs governanceToken, timelock, initialVotingDelay, initialVotingPeriod, initialProposalThreshold, quorumFraction and it passes

- the string "`GovernorOLAS`" to the constructor of `Governor`
- the parameters initialVotingDelay, initialVotingPeriod, initialProposalThreshold to the constructor of `GovernorSettings`
- the parameter governanceToken to the constructor of `GovernorVotes`
- the parameter quorumFraction to the constructor of `GovernorVotesQuorumFraction`
- the parameter timelock to the constructor of `GovernorTimelockControl`

```
    constructor(
        IVotes governanceToken,
        TimelockController timelock,
        uint256 initialVotingDelay,
        uint256 initialVotingPeriod,
        uint256 initialProposalThreshold,
        uint256 quorumFraction
    )
        Governor("GovernorOLAS")
        GovernorSettings(initialVotingDelay, initialVotingPeriod,
initialProposalThreshold)
        GovernorVotes(governanceToken)
        GovernorVotesQuorumFraction(quorumFraction)
        GovernorTimelockControl(timelock)
```

---

feature are included: the abstain vote option, settable parameters (proposal threshold, voting period, voting delay), optional string voting reason, proposer can cancel their proposal (until execution), continuous proposal id logic.

[2] When we use the inheritance of multiple contracts, after the word "is", it is required to write the parents' contracts from the less derived one to the most derived one. However, in Solidity, the resolution order is inverted (for more details consult https://medium.com/coinmonks/inheritance-in-solidity-debunked-3d8dd32d3a99 and https://forum.openzeppelin.com/t/solidity-diamond-inheritance/2694)

```
    {}
```

State

The function `state` is overridden because it is an inherited state from multiple contracts `IGovernor, Governor, GovernorTimelockControl`.

The statement `super.state(proposalId)` at the end will invoke GovernorTimelockControl's original version of the state.

```
   function state(uint256 proposalId) public view override(Governor,
IGovernor, GovernorTimelockControl)
        returns (ProposalState)
    {
        return super.state(proposalId);
    }
```

In particular, `GovernorTimelockControl'` s original version of the state is an overridden version of the Governor function state (the current state of a proposal) with added support for "Queued" status.

Propose

The function `propose` is overridden because it is an inherited state from multiple contracts `IGovernor, Governor, GovernorCompatibilityBravo`.

The statement `super.propose(targets, values, calldatas, description)` at the end will invoke `GovernorCompatibilityBravo`' s original version of the state.

```
    function propose(
        address[] memory targets,
        uint256[] memory values,
        bytes[] memory calldatas,
        string memory description
    ) public override(Governor, GovernorCompatibilityBravo, IGovernor)
returns (uint256)
    {
        return super.propose(targets, values, calldatas, description);
    }
```

In particular, `GovernorCompatibilityBravo`'s original version of the `propose` is an overridden version of the Governor function state (that creates a new proposal) with added a call to the function `storeProposal` that stores a proposal metadata.

Proposal Threshold

The function `proposalThreshold` is overridden because is an inherited state from multiple contract `Governor, GovernorSettings.`

The statement `super.proposalThreshold` at the end will invoke `GovernorSettings`' s original version of the state.

```
    function proposalThreshold() public view override(Governor,
GovernorSettings) returns (uint256)
    {
        return super.proposalThreshold();
    }
```

In particular, `GovernorSettings`'s original version of the `proposalThreshold()` return the number of votes required in order for a voter to become a proposer.

Execute

The function `_execute` is overridden because is an inherited state from multiple contract `Governor, GovernorTimelockControl.`

The statement `super._execute` at the end will invoke `GovernorTimelockControl`' s original version of the state.

```
   function _execute(
        uint256 proposalId,
        address[] memory targets,
        uint256[] memory values,
        bytes[] memory calldatas,
        bytes32 descriptionHash
    ) internal override(Governor, GovernorTimelockControl)
    {
        super._execute(proposalId, targets, values, calldatas,
descriptionHash);
    }
```

In particular, `GovernorTimelockControl’ s` original version of `_execute` is an overridden execute function that run the already queued proposal through the timelock.

Cancel

The function `_cancel` is overridden because is an inherited state from multiple contract `Governor, GovernorTimelockControl.`
The statement `super._cancel` at the end will invoke `GovernorTimelockControl`’ s original version of the state.

```
    function _cancel(
        address[] memory targets,
        uint256[] memory values,
        bytes[] memory calldatas,
        bytes32 descriptionHash
    ) internal override(Governor, GovernorTimelockControl) returns
(uint256)
    {
        return super._cancel(targets, values, calldatas,
descriptionHash);
    }
```

In particular, `GovernorTimelockControl’ s` original version of `_cancel` is an overridden version of the Governor function `_cancel` to cancel the timelocked proposal if it has already been queued.

Executor

The function `_executor` is overridden because is an inherited state from multiple contracts `Governor, GovernorTimelockControl.`
The statement `super._executor` at the end will invoke `GovernorTimelockControl`’ s original version of the state.

```
    function _executor() internal view override(Governor,
GovernorTimelockControl) returns (address)
    {
        return super._executor();
    }
```

In particular, `GovernorTimelockControl’ s` original version of `_executor` return the address of the timelock through which the governor executes action.

SupportsInterface

The function `supportsInterface` is overridden because is an inherited state from multiple contracts `IERC165, Governor, GovernorTimelockControl.`
The statement `super.supportsInterface` at the end will invoke `GovernorTimelockControl`' s original version of the state.

```
    function supportsInterface(bytes4 interfaceId) public view
override(Governor, IERC165, GovernorTimelockControl)
        returns (bool)
    {
        return super.supportsInterface(interfaceId);
    }
```

In particular, `GovernorTimelockControl` inherits from the interface `IERC165` and overrides `supportsInterface` to check for the additional interface Id that are supported.

Contract `VotingEscrow`

Participating in Autononolas governance requires that an account have a balance of veOLAS (locked claim of the OLAS tokens). The veOLAS holder has non-delegatable voting rights. The veOLAS contract is inspired by the Curve DAO VotingEscrow contract. The voting power of each user and the global voting power are recorded in the contract. User voting power is both amounts- and time-weighted, and linearly decreasing since the moment of lock. So does the global voting power. Every time the user deposits, withdraws, or changes the locktime, it is recorded the user's voting power in `mapUserPoints`. And the changes in the global voting power are recorded in `mapSupplyPoints`. In addition, when the user's lock is scheduled to end, there is a schedule change in the slopes of the global vote power in `mapSlopeChanges`. The end time of user locks is rounded off by whole weeks.

Dependencies

The contract `VotingEscrow` inherited the interface IErrors and the OpenZeppelin' interfaces `IVotes`[3], `IERC20`[4], and `IERC165`[5].

Contract-defined data types

It is created the data type `LockedBalance` in the form of a structure for storing balance and unlocking time. The struct contains a group of elements `amount` with data type `int128` and `endTime` with data type `uint64`.

```
struct LockedBalance {
    // Token amount. It will never practically be bigger. Initial OLAS
cap is 1 bn tokens, or 1e27.
    // After 10 years, the inflation rate is 2% per year. It would
take 1340+ years to reach 2^128 - 1
    uint128 amount;
    // Unlock time. It will never practically be bigger
    uint64 endTime;
}
```

It is created the data type `PointVoting` in the form of structure.

```
struct PointVoting {
    // w(i) = at + b (bias)
    int128 bias;
    // dw / dt = a (slope)
    int128 slope;
    // Timestamp. It will never practically be bigger than 2^64 - 1
    uint64 ts;
    // Block number. It will not be bigger than the timestamp
```

[3] Common interface for ERC20Votes. Using this, `VotingEscrow` manteins the compatibility with Tally.
[4] Interface of the ERC20 standard. Using this, `VotingEscrow` manteins the compatibility with Tally.
[5] So `VotingEscrow` contract will inherited form `IERC165` a supportsInterface function that returns:

- true when interfaceID is 0×01ffc9a7 (EIP165 interface)
- false when interfaceID is 0xffffffff
- true for any other interfaceID this contract implements
- false for any other interfaceID

```
    uint64 blockNumber;
    // Token amount. It will never practically be bigger. Initial OLAS
cap is 1 bn tokens, or 1e27.
    // After 10 years, the inflation rate is 2% per year. It would
take 1340+ years to reach 2^128 - 1
    uint128 balance;
}
```

It is created the data type `DepositType` in the form of an enumerated list. The enum contains an enumerated list of elements that are going to be represented with integer values starting from zero.
(e.g. `return DepositType.DEPOSIT_FOR_TYPE;` returns 0
`return DepositType.CREATE_LOCK_TYPE;` returns 1, etc.)

```
   enum DepositType {
        DEPOSIT_FOR_TYPE,
        CREATE_LOCK_TYPE,
        INCREASE_LOCK_AMOUNT,
        INCREASE_UNLOCK_TIME
    }
```

Events

The events `Deposit`, `Withdraw`, and `Supply` are declared.

```
    event Deposit(address provider, uint256 amount, uint256 locktime,
DepositType depositType, uint256 ts);
    event Withdraw(address indexed provider, uint256 amount, uint256
ts);
    event Supply(uint256 prevSupply, uint256 supply);
```

Variables

The internal constant `WEEK` is 7 days.

```
    uint256 internal constant WEEK = 1 weeks;
```

The internal constant `MAXTIME` sets the maximum locking time at 4 years calculated in seconds

```
    uint256 internal constant MAXTIME = 4 * 365 * 86400;
```

The `token` is a public and immutable variable that indicates the token address

```
    address immutable public token;
```

The `supply` is a public variable that indicates the total token supply

```
    uint256 public supply;
```

The `totalNumPoints` is a public variable that indicates the total number of checkpoints

```
    uint256 public totalNumPoints;
```

Mappings

The public mapping `mapLockedBalances` maps the user's `address` to his corresponding `LockedBalance`

```
    mapping(address => LockedBalance) public mapLockedBalances;
```

The public mapping `mapSupplyPoints` maps a point to the corresponding `PointVoting`

```
    // Mapping of point Id => point
    mapping(uint256 => PointVoting) public mapSupplyPoints;
```

The public mapping `mapUserPoints` maps the user's `address` to his corresponding `PointVoting` at various points

```
    // Mapping of account address => PointVoting[point Id]
    mapping(address => PointVoting[]) public mapUserPoints;
```

The public mapping `mapSlopeChanges` maps a time to its corresponding slope change

```
    // Mapping of time => signed slope change
```

```
    mapping(uint256 => int128) public mapSlopeChanges;
```

Constructor

Sets the values for the `address`, `name`, and `symbol`. The initial point is at `block.timestamp` and at `block.number` and, at that point, the `bias`, the `slope`, and the `balance` are 0.

```
    constructor(address _token, string memory _name, string memory
_symbol)
    {
        token = _token;
        name = _name;
        symbol = _symbol;
        decimals = ERC20(_token).decimals();
        // Create initial point such that default timestamp and block
number are not zero
        mapSupplyPoints[0] = PointVoting(0, 0, block.timestamp,
block.number, 0);
    }
```

getLastUserPoint

The external function `getLastUserPoint(address account)`, when there are at least two `PointVoting` recorded for the user with the address `account`, returns the most recent `PointVoting` recorded for that user

```
    function getLastUserPoint(address account) external view returns
(PointVoting memory pv) {
        uint256 lastPointNumber = mapUserPoints[account].length;
        if (lastPointNumber > 0) {
            pv = mapUserPoints[account][lastPointNumber - 1];
        }
   }
```

getNumUserPoint

The external function `getNumUserPoints(address account)` returns the number of `PointVoting` recorded for the user with the address `account`

```
    function getNumUserPoints(address account) external view returns
(uint256 accountNumPoints) {
        accountNumPoints = mapUserPoints[account].length;
    }
```

getUserPoint

The external function `getUserPoints(address account, uint256 idx)` returns the `PointVoting` at the point with the number `idx` recorded for the user with the address `account`.

```
    function getUserPoint(address account, uint256 idx) external view
returns (PointVoting memory) {
        return mapUserPoints[account][idx];
    }
```

_checkpoint

The function `_checkpoint(address account, LockedBalance memory oldLocked, LockedBalance memory newLocked)` is internal to the contract and its inputs parameters are the user's address `account`, the user's precedent locked amount - the user's precedent end of the lock time (`oldLocked)`, the user's new locked amount - the user's new end of lock time (`newLocked)`. The function records the users (with non-zero addresses) data and global data to the checkpoint.

```
      function _checkpoint(
        address account,
        LockedBalance memory oldLocked,
        LockedBalance memory newLocked,
        uint128 curSupply
    ) internal {
        PointVoting memory uOld;
        PointVoting memory uNew;
        int128 oldDSlope;
        int128 newDSlope;
        uint256 curNumPoint = totalNumPoints;
```

```
        if (account != address(0)) {
            // Calculate slopes and biases
            // Kept at zero when they have to
            if (oldLocked.endTime > block.timestamp &&
oldLocked.amount > 0) {
                uOld.slope = int128(oldLocked.amount) / IMAXTIME;
                uOld.bias = uOld.slope *
int128(uint128(oldLocked.endTime - uint64(block.timestamp)));
            }
            if (newLocked.endTime > block.timestamp &&
newLocked.amount > 0) {
                uNew.slope = int128(newLocked.amount) / IMAXTIME;
                uNew.bias = uNew.slope *
int128(uint128(newLocked.endTime - uint64(block.timestamp)));
            }


            // Reads values of scheduled changes in the slope
            // oldLocked.endTime can be in the past and in the future
            // newLocked.endTime can ONLY be in the FUTURE unless
everything is expired: then zeros
            oldDSlope = mapSlopeChanges[oldLocked.endTime];
            if (newLocked.endTime > 0) {
                if (newLocked.endTime == oldLocked.endTime) {
                    newDSlope = oldDSlope;
                } else {
                    newDSlope = mapSlopeChanges[newLocked.endTime];
                }
            }
        }


        PointVoting memory lastPoint;
        if (curNumPoint > 0) {
            lastPoint = mapSupplyPoints[curNumPoint];
        } else {
            // If no point is created yet, we take the actual time and
block parameters
            lastPoint = PointVoting(0, 0, uint64(block.timestamp),
uint64(block.number), 0);
        }
        uint64 lastCheckpoint = lastPoint.ts;
        // initialPoint is used for extrapolation to calculate the
block number and save them
```

```
        // as we cannot figure that out in exact values from inside of
the contract
        PointVoting memory initialPoint = lastPoint;
        uint256 block_slope; // dblock/dt
        if (block.timestamp > lastPoint.ts) {
            // This 1e18 multiplier is needed for the numerator to be
bigger than the denominator
            // We need to calculate this in > uint64 size (1e18 is >
2^59 multiplied by 2^64).
            block_slope = (1e18 * uint256(block.number -
lastPoint.blockNumber)) / uint256(block.timestamp - lastPoint.ts);
        }
        // If last point is already recorded in this block, slope ==
0, but we know the block already in this case
        // Go over weeks to fill in the history and (or) calculate
what the current point is
        {
            // The timestamp is rounded and < 2^64-1
            uint64 tStep = (lastCheckpoint / WEEK) * WEEK;
            for (uint256 i = 0; i < 255; ++i) {
                // Hopefully it won't happen that this won't get used
in 5 years!
                // If it does, users will be able to withdraw but vote
weight will be broken
                // This is always practically < 2^64-1
                unchecked {
                    tStep += WEEK;
                }
                int128 dSlope;
                if (tStep > block.timestamp) {
                    tStep = uint64(block.timestamp);
                } else {
                    dSlope = mapSlopeChanges[tStep];
                }
                lastPoint.bias -= lastPoint.slope * int128(int64(tStep
- lastCheckpoint));
                lastPoint.slope += dSlope;
                if (lastPoint.bias < 0) {
                    // This could potentially happen, but fuzzer
didn't find available "real" combinations
                    lastPoint.bias = 0;
                }
                if (lastPoint.slope < 0) {
                    // This cannot happen - just in case. Again,
fuzzer didn't reach this
```

```
                    lastPoint.slope = 0;
                }
                lastCheckpoint = tStep;
                lastPoint.ts = tStep;
                // After division by 1e18 the uint64 size can be
reclaimed
                lastPoint.blockNumber = initialPoint.blockNumber +
uint64((block_slope * uint256(tStep - initialPoint.ts)) / 1e18);
                lastPoint.balance = initialPoint.balance;
                // In order for the overflow of total number of
economical checkpoints (starting from zero)
                // The _checkpoint() call must happen n >(2^256
-1)/255 or n > ~1e77/255 > ~1e74 times
                unchecked {
                    curNumPoint += 1;
                }
                if (tStep == block.timestamp) {
                    lastPoint.blockNumber = uint64(block.number);
                    lastPoint.balance = curSupply;
                    break;
                } else {
                    mapSupplyPoints[curNumPoint] = lastPoint;
                }
            }
        }


        totalNumPoints = curNumPoint;


        // Now mapSupplyPoints is filled until current time
        if (account != address(0)) {
            // If last point was in this block, the slope change has
been already applied. In such case we have 0 slope(s)
            lastPoint.slope += (uNew.slope - uOld.slope);
            lastPoint.bias += (uNew.bias - uOld.bias);
            if (lastPoint.slope < 0) {
                lastPoint.slope = 0;
            }
            if (lastPoint.bias < 0) {
                lastPoint.bias = 0;
            }
        }
```

```
        // Record the last updated point
        mapSupplyPoints[curNumPoint] = lastPoint;


        if (account != address(0)) {
            // Schedule the slope changes (slope is going down)
            // We subtract new_user_slope from [newLocked.endTime]
            // and add old_user_slope to [oldLocked.endTime]
            if (oldLocked.endTime > block.timestamp) {
                // oldDSlope was <something> - uOld.slope, so we
cancel that
                oldDSlope += uOld.slope;
                if (newLocked.endTime == oldLocked.endTime) {
                    oldDSlope -= uNew.slope; // It was a new deposit,
not extension
                }
                mapSlopeChanges[oldLocked.endTime] = oldDSlope;
            }


            if (newLocked.endTime > block.timestamp &&
newLocked.endTime > oldLocked.endTime) {
                newDSlope -= uNew.slope; // old slope disappeared at
this point
                mapSlopeChanges[newLocked.endTime] = newDSlope;
                // else: we recorded it already in oldDSlope
            }
            // Now handle user history
            uNew.ts = uint64(block.timestamp);
            uNew.blockNumber = uint64(block.number);
            uNew.balance = newLocked.amount;
            mapUserPoints[account].push(uNew);
        }
    }
```

checkpoint

The external `checkpoint()` records the global data to the checkpoint

```
    function checkpoint() external {
        _checkpoint(address(0), LockedBalance(0, 0), LockedBalance(0,
0), uint128(supply));
```

```
    }
```

_depositFor

The internal function `_depositFor(address account, uint256 amount, uint256 unlockTime, LockedBalance memory lockedBalance, DepositType depositType )` has input the address `account` to which deposit, the `amount` to deposit, the existing `lockedBalance` of the address `account`, and the deposit type.

The function gets the old locked data and either creates a new lock or, when there is already a locked amount for the `account`, the `amount` is added to the previously locked one. If `unlockTime` is non-zero, `unlockTime` becomes the end of the locking, otherwise, the end of the locking is the previous unlocking time. The function calls the checkpoint to register the user's address `account` and global data, the `oldLocked`, and the `lockedBalance`. Whether the `amount` is non-zero, an `IERC20(token).transferFrom(msg.sender, address(this), amount)` of the `amount` from the user's `account`, to this `VotingEsrow` is made. It follows an emission of the following events

```
Deposit(account, amount, lockedBalance.end, depositType,
block.timestamp);
Supply(supplyBefore, supplyBefore + amount);
```

```
       function _depositFor(
        address account,
        uint256 amount,
        uint256 unlockTime,
        LockedBalance memory lockedBalance,
        DepositType depositType
    ) internal {
        uint256 supplyBefore = supply;
        uint256 supplyAfter;
        // Cannot overflow because the total supply << 2^128-1
        unchecked {
            supplyAfter = supplyBefore + amount;
            supply = supplyAfter;
        }
        // Get the old locked data
        LockedBalance memory oldLocked;
        (oldLocked.amount, oldLocked.endTime) = (lockedBalance.amount,
lockedBalance.endTime);
```

```
        // Adding to the existing lock, or if a lock is expired -
creating a new one
        // This cannot be larger than the total supply
        unchecked {
            lockedBalance.amount += uint128(amount);
        }
        if (unlockTime > 0) {
            lockedBalance.endTime = uint64(unlockTime);
        }
        mapLockedBalances[account] = lockedBalance;


        // Possibilities:
        // Both oldLocked.endTime could be current or expired (>/<
block.timestamp)
        // amount == 0 (extend lock) or amount > 0 (add to lock or
extend lock)
        // lockedBalance.endTime > block.timestamp (always)
        _checkpoint(account, oldLocked, lockedBalance,
uint128(supplyAfter));
        if (amount > 0) {
            // OLAS is a solmate-based ERC20 token with optimized
transferFrom() that either returns true or reverts
            IERC20(token).transferFrom(msg.sender, address(this),
amount);
        }


        emit Deposit(account, amount, lockedBalance.endTime,
depositType, block.timestamp);
        emit Supply(supplyBefore, supplyAfter);
    }
```

depositFor

The function `depositFor(address account, uint256 amount)` is an external function. Whether the `amount` is non-zero, the `account` has already a locked amount that is not expired yet, it is made a call to the function `_depositFor(account, amount, 0, lockedBalance, DepositType.DEPOSIT_FOR_TYPE)` that increases the amount of tokens of the `account` but not extending `account's` unlocking time. Since in `_depositFor()` there is an unchecked sum of amounts, priory to the call of the function

`_depositFor()is is checked that amount < type(uint96).max` prior otherwise the request is reverted.

```
        function depositFor(address account, uint256 amount) external
{
        LockedBalance memory lockedBalance =
mapLockedBalances[account];
        // Check if the amount is zero
        if (amount == 0) {
            revert ZeroValue();
        }
        // The locked balance must already exist
        if (lockedBalance.amount == 0) {
            revert NoValueLocked(account);
        }
        // Check the lock expiry
        if (lockedBalance.endTime < (block.timestamp + 1)) {
            revert LockExpired(msg.sender, lockedBalance.endTime,
block.timestamp);
        }
        // Since in the _depositFor() we have the unchecked sum of
amounts, this is needed to prevent unsafe behavior.
        // After 10 years, the inflation rate is 2% per year. It would
take 220+ years to reach 2^96 - 1 total supply
        if (amount > type(uint96).max) {
            revert Overflow(amount, type(uint96).max);
        }


        _depositFor(account, amount, 0, lockedBalance,
DepositType.DEPOSIT_FOR_TYPE);
    }

    }
```

createLock

The function `createLock(uint256 amount, uint256 unlockTime)` is an external function and it is made a call to the function `_createLockFor(msg.sender, amount, unlockTime)` that creates a new locking for the `msg.sender` of the `amount` for `unlockTime`.

```
    /// @dev Deposits `amount` tokens for `msg.sender` and locks for
`unlockTime`.
    /// @param amount Amount to deposit.
    /// @param unlockTime Time when tokens unlock, rounded down to a
whole week.
    function createLock(uint256 amount, uint256 unlockTime) external {
        _createLockFor(msg.sender, amount, unlockTime);
    }
```

createLockFor

The function `createLockFor(address account, uint256 amount, uint256 unlockTime)` is an external function that allows anyone to deposit the `amount` of tokens for a (non-zero) `account` and lock those for `unlockTime` via the function `_createLockFor(account, amount, unlockTime)`.

```
    /// @dev Deposits `amount` tokens for `account` and locks for
`unlockTime`.
    /// @notice Tokens are taken from `msg.sender`'s balance.
    /// @param account Account address.
    /// @param amount Amount to deposit.
    /// @param unlockTime Time when tokens unlock, rounded down to a
whole week.
    function createLockFor(address account, uint256 amount, uint256
unlockTime) external {
        // Check if the account address is zero
        if (account == address(0)) {
            revert ZeroAddress();
        }


        _createLockFor(account, amount, unlockTime);
    }
```

_createLock

The function `_createLock(uint256 amount, uint256 amount, uint256 unlockTime)`. Whether the `amount` is non-zero and not bigger than 2^96 - 1, the `account` has zero locked amount, the `unlockTime` is bigger than the `block.timestamp+1`, and smaller than the maximum allowed unlockeTime (i.e. 4 years),

it is made a call to the function `_depositFor(account, amount, unlockTime, lockedBalance, DepositType.CREATE_LOCK_TYPE)` that creates a new locking for the `account` of the `amount` for `unlockTime`.

```
    /// @dev Deposits `amount` tokens for `account` and locks for
`unlockTime`.
    /// @notice Tokens are taken from `msg.sender`'s balance.
    /// @param account Account address.
    /// @param amount Amount to deposit.
    /// @param unlockTime Time when tokens unlock, rounded down to a
whole week.
    function _createLockFor(address account, uint256 amount, uint256
unlockTime) private {
        // Check if the amount is zero
        if (amount == 0) {
            revert ZeroValue();
        }
        // Lock time is rounded down to weeks
        // Cannot practically overflow because block.timestamp +
unlockTime (max 4 years) << 2^64-1
        unchecked {
            unlockTime = ((block.timestamp + unlockTime) / WEEK) *
WEEK;
        }
        LockedBalance memory lockedBalance =
mapLockedBalances[account];
        // The locked balance must be zero in order to start the lock
        if (lockedBalance.amount > 0) {
            revert LockedValueNotZero(account,
uint256(lockedBalance.amount));
        }
        // Check for the lock time correctness
        if (unlockTime < (block.timestamp + 1)) {
            revert UnlockTimeIncorrect(account, block.timestamp,
unlockTime);
        }
        // Check for the lock time not to exceed the MAXTIME
        if (unlockTime > block.timestamp + MAXTIME) {
            revert MaxUnlockTimeReached(account, block.timestamp +
MAXTIME, unlockTime);
        }
        // After 10 years, the inflation rate is 2% per year. It would
take 220+ years to reach 2^96 - 1 total supply
```

```
        if (amount > type(uint96).max) {
            revert Overflow(amount, type(uint96).max);
        }


        _depositFor(account, amount, unlockTime, lockedBalance,
DepositType.CREATE_LOCK_TYPE);
    }
```

increaseAmount

The function `increaseAmount(uint256 amount)` is an external function. Whether the `amount` is non-zero and not bigger than 2^96 - 1, the `msg.sender` calling the function has already a locked amount that is scheduled to unlock 'later' the call of this function, it is made a call to the function `_depositFor(msg.sender, amount, 0, lockedBalance, DepositType.INCREASE_LOCK_AMOUNT)` that increase the locked amount for the `msg.sender` of the `amount` without modifying the unlock time.

```
    function increaseAmount(uint256 amount) external {
        LockedBalance memory lockedBalance =
mapLockedBalances[msg.sender];
        // Check if the amount is zero
        if (amount == 0) {
            revert ZeroValue();
        }
        // The locked balance must already exist
        if (lockedBalance.amount == 0) {
            revert NoValueLocked(msg.sender);
        }
        // Check the lock expiry
        if (lockedBalance.endTime < (block.timestamp + 1)) {
            revert LockExpired(msg.sender, lockedBalance.endTime,
block.timestamp);
        }
        // Check the max possible amount to add, that must be less
than the total supply
        // After 10 years, the inflation rate is 2% per year. It would
take 220+ years to reach 2^96 - 1 total supply
```

```
        if (amount > type(uint96).max) {
            revert Overflow(amount, type(uint96).max);
        }


        _depositFor(msg.sender, amount, 0, lockedBalance,
DepositType.INCREASE_LOCK_AMOUNT);
    }
```

increaseUnlockTime

The function `increaseUnlockTime(uint256 unlockTime)` is an external function. Whether the locked amount of the `msg.sender` calling the function is non-zero, the `unlockTime` is bigger than the old unlock time of the `msg.sender plus one`, and the `unlockTime` does not exceed the maximum time to unlock, then a `_depositFor(msg.sender , amount, 0, lockedBalance, DepositType.INCREASE_UNLOCK_TIME)` increases the `unlockTime` for the `msg.sender` without modifying its locked amount.

```
        function increaseUnlockTime(uint256 unlockTime) external {
        LockedBalance memory lockedBalance =
mapLockedBalances[msg.sender];
        // Cannot practically overflow because block.timestamp +
unlockTime (max 4 years) << 2^64-1
        unchecked {
            unlockTime = ((block.timestamp + unlockTime) / WEEK) *
WEEK;
        }
        // The locked balance must already exist
        if (lockedBalance.amount == 0) {
            revert NoValueLocked(msg.sender);
        }
        // Check the lock expiry
        if (lockedBalance.endTime < (block.timestamp + 1)) {
            revert LockExpired(msg.sender, lockedBalance.endTime,
block.timestamp);
        }
        // Check for the lock time correctness
        if (unlockTime < (lockedBalance.endTime + 1)) {
            revert UnlockTimeIncorrect(msg.sender,
lockedBalance.endTime, unlockTime);
        }
```

```
        // Check for the lock time not to exceed the MAXTIME
        if (unlockTime > block.timestamp + MAXTIME) {
            revert MaxUnlockTimeReached(msg.sender, block.timestamp +
MAXTIME, unlockTime);
          }


        _depositFor(msg.sender, 0, unlockTime, lockedBalance,
DepositType.INCREASE_UNLOCK_TIME);
    }
```

withdraw

The function `withdraw()` is an external function. Whether the old locked balance of the `msg.sender` has expired (at least at the `block.timestamp` of the function call) the function call by the `msg.sender`, the user-data are recorded at the checkpoint where the old amount/unlock time of `msg.sende` are fixed at zero, a `IERC20(token).transfer(msg.sender, amount)` transfers all the tokens of the `msg.sender` to its account. The following events are emitted

```
  emit Withdraw(msg.sender, amount, block.timestamp);
  emit Supply(supplyBefore, supplyAfter);
```

```
    function withdraw() external {
        LockedBalance memory lockedBalance =
mapLockedBalances[msg.sender];
        if (lockedBalance.endTime > block.timestamp) {
            revert LockNotExpired(msg.sender, lockedBalance.endTime,
block.timestamp);
        }
        uint256 amount = uint256(lockedBalance.amount);


        mapLockedBalances[msg.sender] = LockedBalance(0, 0);
        uint256 supplyBefore = supply;
        uint256 supplyAfter;
        // The amount cannot be less than the total supply
        unchecked {
            supplyAfter = supplyBefore - amount;
```

```
            supply = supplyAfter;
        }
        // oldLocked can have either expired <= timestamp or zero end
        // lockedBalance has only 0 end
        // Both can have >= 0 amount
        _checkpoint(msg.sender, lockedBalance, LockedBalance(0, 0),
uint128(supplyAfter));


        emit Withdraw(msg.sender, amount, block.timestamp);
        emit Supply(supplyBefore, supplyAfter);


        // OLAS is a solmate-based ERC20 token with optimized
transfer() that either returns true or reverts
        IERC20(token).transfer(msg.sender, amount);
    }
```

_findPointByBlock

The function `_findPointByBlock(uint256 blockNumber, address account)` is an internal function that finds the closest point with a specific `blockNumber`. This function returns the `PointVoting` with the index closest to the point for the specified `blockNumber` and the `PointVoting` number.

```
    function _findPointByBlock(uint256 blockNumber, address account)
internal view
        returns (PointVoting memory point, uint256 minPointNumber)
    {
        // Get the last available point number
        uint256 maxPointNumber;
        if (account == address(0)) {
            maxPointNumber = totalNumPoints;
        } else {
            maxPointNumber = mapUserPoints[account].length;
            if (maxPointNumber == 0) {
                return (point, minPointNumber);
            }
            // Already checked for > 0 in this case
```

```
            unchecked {
                maxPointNumber -= 1;
            }
        }


        // Binary search that will be always enough for 128-bit
numbers
        for (uint256 i = 0; i < 128; ++i) {
            if ((minPointNumber + 1) > maxPointNumber) {
                break;
            }
            uint256 mid = (minPointNumber + maxPointNumber + 1) / 2;


            // Choose the source of points
            if (account == address(0)) {
                point = mapSupplyPoints[mid];
            } else {
                point = mapUserPoints[account][mid];
            }


            if (point.blockNumber < (blockNumber + 1)) {
                minPointNumber = mid;
            } else {
                maxPointNumber = mid - 1;
            }
        }


        // Get the found point
        if (account == address(0)) {
            point = mapSupplyPoints[minPointNumber];
        } else {
            point = mapUserPoints[account][minPointNumber];
        }
    }
```

_balanceOfLocked

The function `_balanceOfLocked(address account, uint256 ts)` is an internal function that returns the voting power of a user with an address `account` at the block with timestamp `ts`.

```
    function _balanceOfLocked(address account, uint64 ts) internal
view returns (uint256 vBalance) {
        uint256 pointNumber = mapUserPoints[account].length;
        if (pointNumber > 0) {
            PointVoting memory uPoint =
mapUserPoints[account][pointNumber - 1];
            uPoint.bias -= uPoint.slope * int128(int64(ts) -
int64(uPoint.ts));
            if (uPoint.bias > 0) {
                vBalance = uint256(int256(uPoint.bias));
            }
        }
    }
```

balanceOf

The function `function balanceOf(address account)` is an public function.
It is marked as override because the contract `VotingEscrow` calls the interface `OpenZeppelin - IERC20` containing the function "`function balanceOf(address account)`".
This function simply returns the amount of tokens that the user with the address `account` has.

```
    function balanceOf(address account) public view override returns
(uint256 balance) {
        balance = uint256(mapLockedBalances[account].amount);
    }
```

lockendEnd

The function `lockedEnd(address account)` is an external function. This function returns the scheduled end of the locking time for the user with the address `account`.

```
    function lockedEnd(address account) external view returns (uint256
unlockTime) {
        unlockTime = uint256(mapLockedBalances[account].endTime);
    }
```

balanceOfAt

The function `function balanceOfAt(address account, uint256 blockNumber)` is an external function. This function gets the account balance at a specific block number.

```
    function balanceOfAt(address account, uint256 blockNumber)
external view returns (uint256 balance) {
        // Find point with the closest block number to the provided
one
        (PointVoting memory uPoint, ) = _findPointByBlock(blockNumber,
account);
        // If the block number at the point index is bigger than the
specified block number, the balance was zero
        if (uPoint.blockNumber < (blockNumber + 1)) {
            balance = uint256(uPoint.balance);
        }
    }
```

getVotes

The function `getVotes(address account)` is a public function. It is marked as override because the contract `VotingEscrow` calls the interface `OpenZeppelin - IVotes` containing the function "`function getVotes(address account)`". This function, when it is called, returns the voting power of the user with the address `account`.

```
    function getVotes(address account) public view override returns
(uint256) {
        return _balanceOfLocked(account, uint64(block.timestamp));
    }
```

_getBlockTime

The function `_getBlockTime(uint256 blockNumber)` is an internal function. Whether the `blockNumber` is smaller or equal to the block number corresponding to when such a function gets called, then the function returns the point with the specified `blockNumber` (or closest to it) and the adjusted block time of the neighboring point.

```
    function _getBlockTime(uint256 blockNumber) internal view returns
(PointVoting memory point, uint256 blockTime) {
        // Check the block number to be in the past or equal to the
current block
        if (blockNumber > block.number) {
            revert WrongBlockNumber(blockNumber, block.number);
        }
        // Get the minimum historical point with the provided block
number
        uint256 minPointNumber;
        (point, minPointNumber) = _findPointByBlock(blockNumber,
address(0));


        uint256 dBlock;
        uint256 dt;
        if (minPointNumber < totalNumPoints) {
            PointVoting memory pointNext =
mapSupplyPoints[minPointNumber + 1];
            dBlock = pointNext.blockNumber - point.blockNumber;
            dt = pointNext.ts - point.ts;
        } else {
            dBlock = block.number - point.blockNumber;
            dt = block.timestamp - point.ts;
        }
        blockTime = point.ts;
        if (dBlock > 0) {
            blockTime += (dt * (blockNumber - point.blockNumber)) /
dBlock;
        }
    }
```

getPastVotes

The function `getPastVotes(address account, uint256 blockNumber)` is a public function. It is marked as override because the contract `VotingEscrow` calls the interface `OpenZeppelin - IVotes` containing the function "`function getPastVotes(address account, uint256 blockNumber)`".
Whether the voting power is non-zero, the function returns the voting power of the `account` at `blockNumber`.

```
       function getPastVotes(address account, uint256 blockNumber)
public view override returns (uint256 balance) {
        // Find the user point for the provided block number
        (PointVoting memory uPoint, ) = _findPointByBlock(blockNumber,
account);


        // Get block time adjustment.
        (, uint256 blockTime) = _getBlockTime(blockNumber);


        // Calculate bias based on a block time
        uPoint.bias -= uPoint.slope * int128(int64(uint64(blockTime))
- int64(uPoint.ts));
        if (uPoint.bias > 0) {
            balance = uint256(uint128(uPoint.bias));
        }
    }
```

_supplyLockedAt

The function `_supplyLockedAt(PointVoting memory lastPoint, uint256 ts)` is an internal function. The input `lastPoint` is the starting point to start the search for calculating the global voting power at the time `ts`. The function returns the global voting power when the latter is bigger than zero.

```
    function _supplyLockedAt(PointVoting memory lastPoint, uint64 ts)
internal view returns (uint256 vSupply) {
        // The timestamp is rounded and < 2^64-1
        uint64 tStep = (lastPoint.ts / WEEK) * WEEK;
        for (uint256 i = 0; i < 255; ++i) {
            // This is always practically < 2^64-1
```

```
            unchecked {
                tStep += WEEK;
            }
            int128 dSlope;
            if (tStep > ts) {
                tStep = ts;
            } else {
                dSlope = mapSlopeChanges[tStep];
            }
            lastPoint.bias -= lastPoint.slope * int128(int64(tStep) -
int64(lastPoint.ts));
            if (tStep == ts) {
                break;
            }
            lastPoint.slope += dSlope;
            lastPoint.ts = tStep;
        }


        if (lastPoint.bias > 0) {
            vSupply = uint256(uint128(lastPoint.bias));
        }
    }
```

totalSypply

The function `totalSupply()` is a public function that returns the total token supply. It is marked as override because the contract VotingEscrow calls the interface IERC20 and the latter contains the function "`totalSupply()`".

```
    function totalSupply() public view override returns (uint256) {
        return supply;
    }
```

totalSypplyAt

The function `totalSupplyAt(uint256 blockNumber)` is an external function. If the block number of the last point closest to `blockNumber` is smaller or equal to `blockNumber` itself, then the function returns the global voting power at the specific `blockNumber`.

```
    function totalSupplyAt(uint256 blockNumber) external view returns
(uint256 supplyAt) {
        // Find point with the closest block number to the provided
one
        (PointVoting memory sPoint, ) = _findPointByBlock(blockNumber,
address(0));
        // If the block number at the point index is bigger than the
specified block number, the balance was zero
        if (sPoint.blockNumber < (blockNumber + 1)) {
            supplyAt = uint256(sPoint.balance);
        }
    }
```

totalSypplyLockedAtT

The function `totalSupplyLockedAtT(uint256 ts)` is a public function and it calculates the global voting power at the specific time `ts`.

```
       function totalSupplyLockedAtT(uint256 ts) public view returns
(uint256) {
        PointVoting memory lastPoint =
mapSupplyPoints[totalNumPoints];
        return _supplyLockedAt(lastPoint, uint64(ts));
    }
```

totalSypplyLocked

The function `totalSupplyLocked()` is a public function that returns the current global voting power.

```
       function totalSupplyLocked() public view returns (uint256) {
        return totalSupplyLockedAtT(block.timestamp);
```

```
    }
```

getPastTotalSupply

The function `getPastTotalSupply(uint256 blockNumber)` is a public function that returns the global voting power at the specific `blockNumber`. It is marked as override because the contract `VotingEscrow` calls the interfaces `IVotes` that contains the function "`totalSupply()`".

```
     function getPastTotalSupply(uint256 blockNumber) public view
override returns (uint256) {
        (PointVoting memory sPoint, uint256 blockTime) =
_getBlockTime(blockNumber);
        // Now dt contains info on how far are we beyond the point
        return _supplyLockedAt(sPoint, uint64(blockTime));
    }
```

supportInterface

The function `supportsInterface(bytes4 interfaceId)` is a public function that gets information about additional interface-id that are supported by the contract. It is marked as override because the contract calls the interface `IERC165` that contains the function "`supportsInterface`".

```
    function supportsInterface(bytes4 interfaceId) public view virtual
override returns (bool) {
        return interfaceId == type(IERC20).interfaceId || interfaceId
== type(IVotes).interfaceId ||
            interfaceId == type(IERC165).interfaceId;
    }
```

Transfer

The function `transfer(address to, uint256 amount)` is a public function and it is marked as override because the contract `VotingEscrow` calls the interfaces `IERC20` that

contains the function "`transfer(address to, uint256 amount)`". This function reverts any transfer of any amount to an address.

```
    function transfer(address to, uint256 amount) external
  virtual override returns (bool) {
        revert NonTransferable(address(this));
    }
```

Approve

The function `approve(address spender, uint256 amount)` is a public function and it is marked as override because the contract `VotingEscrow` calls the interfaces `IERC20` that contains the function "`approve(address spender, uint256 amount)`". This function reverts any approval of the spent amount from an address spender

```
    function approve(address spender, uint256 amount) external virtual
override returns (bool) {
        revert NonTransferable(address(this));
    }
```

TransferFrom

The function `transferFrom(address from, address to, uint256 amount)` is a public function and it is marked as override because the contract `VotingEscrow` calls the interfaces `IERC20`. This function reverts transfer from an address to an address of any amount.

```
    function transferFrom(address from, address to, uint256 amount)
external virtual override returns (bool) {
        revert NonTransferable(address(this));
    }
```

Allowance

The function `allowance(address owner, address spender)` is a public function and it is marked as override because the contract `VotingEscrow` calls the interfaces

IERC20. This function reverts any change of allowance of the remaining tokens that the spender would be allowed to spend on the owner's behalf if approve or transferFrom were called

```
    function allowance(address owner, address spender) external view
virtual override returns (uint256)
    {
        revert NonTransferable(address(this));
    }
```

Delegates

The function `delegates(address account)` is a public function and it is marked as override because the contract VotingEscrow calls the interfaces IVotes. This function reverts any address to which an account would have chosen to delegate.

```
    function delegates(address account) external view virtual override
returns (address)
    {
        revert NonDelegatable(address(this));
    }
```

Delegate

The function `delegate(address delegatee)` is a public function and it is marked as override because the contract VotingEscrow calls the interfaces IVotes. This function reverts any delegate of votes from the sender

```
    function delegate(address delegatee) external virtual override
    {
        revert NonDelegatable(address(this));
    }
```

DelegateBySig

The function `delegateBySig(address delegatee, uint256 nonce, uint256 expiry, uint8 v, bytes32 r, bytes32 s)` is a public function and it is marked as override because the contract `VotingEscrow` calls the interfaces `IVotes`. This function reverts any delegate votes from a signer.

```
    function delegateBySig(address delegatee, uint256 nonce, uint256
expiry, uint8 v, bytes32 r, bytes32 s)
        external virtual override
    {
        revert NonDelegatable(address(this));
    }
}
```

Contract `buOLAS`

Contract-defined Interface

The interface IOLAS is an interface for burn functionality.

```
interface IOLAS {
    /// @dev Burns OLAS tokens.
    /// @param amount OLAS token amount to burn.
    function burn(uint256 amount) external;
}
```

Contract-defined data types

It is created the data type `LockedBalance` in the form of a structure for storing balance and unlocking time. The struct contains a group of elements: `totalAmount` with data type `uint96`, `transferredAmount` with data type `uint96`, `startTime` with data type `uint32`, `endTime` with data type `uint32`.

```
struct LockedBalance {
```

```
    // Token amount locked. Initial OLAS cap is 1 bn tokens, or 1e27.
    // After 10 years, the inflation rate is 2% per year. It would
take 220+ years to reach 2^96 - 1
    uint96 totalAmount;
    // Token amount transferred to its owner. It is of the value of at
most the total amount locked
    uint96 transferredAmount;
    // Lock time start
    uint32 startTime;
    // Lock end time
    // 2^32 - 1 is enough to count 136 years starting from the year of
1970. This counter is safe until the year of 2106
    uint32 endTime;
}
```

Dependencies

The contract `buOLAS` inherited the interface `IErrors` and the OpenZeppelin' interfaces `IERC20`[6] and `IERC165`[7].

Events

The events `Lock`, `Withdraw`, `Revoke`, `Burn`, `Supply`, and `OwnerUpdated` are declared.

```
 event Lock(address indexed account, uint256 amount, uint256
startTime, uint256 endTime);
    event Withdraw(address indexed account, uint256 amount, uint256
ts);
    event Revoke(address indexed account, uint256 amount, uint256 ts);
    event Burn(address indexed account, uint256 amount, uint256 ts);
    event Supply(uint256 previousSupply, uint256 currentSupply);
    event OwnerUpdated(address indexed owner);
```

[6] Interface of the ERC20 standard.
[7] So `buOLAS` contract will inherited form `IERC165` a supportsInterface function.

Variables

The internal constant `STEP_TIME` fixes the step time calculated in seconds

```
uint32 internal constant STEP_TIME = 365 * 86400;
```

The internal constant `MAX_NUM_STEPS` fixes the maximal number of steps for locking calculated in seconds

```
uint32 internal constant MAX_NUM_STEPS = 10;
```

The `supply` is a public variable that indicates the total token supply

```
uint256 public supply;
```

The `decimal` is a public variable that set 18 decimals

```
uint8 public constant decimals = 18;
```

The `token` is a public and immutable variable that indicates the token address

```
address public immutable token;
```

The `owner` is a public variable that indicates the owner's address

```
address public owner;
```

Mappings

The public mapping `mapLockedBalances` maps the user's `address` to his corresponding `LockedBalance`

```
mapping(address => LockedBalance) public mapLockedBalances;
```

Constructor

Sets the values for the `address`, `name`, `symbol`, and makes the contract ownable with `owner = msg.sender`.

```
    constructor(address _token, string memory _name, string memory
_symbol)
        {
         token = _token;
         name = _name;
         symbol = _symbol;
         owner = msg.sender;
    }
```

ChangeOwner

The function `changeOwner(address newOwner)` is an external function that allows the previous owner of the contract to sets a new owner (with a no-zero address) and emit the event

```
 emit OwnerUpdated(newOwner);
```

```
    function changeOwner(address newOwner) external {
        if (msg.sender != owner) {
            revert OwnerOnly(msg.sender, owner);
        }


        if (newOwner == address(0)) {
            revert ZeroAddress();
        }


        owner = newOwner;
        emit OwnerUpdated(newOwner);
    }
```

CreateLockFor

The function `createLockFor(address account, uint256 amount, uint256 numSteps)` is an external function that allows depositing to the address `account` the

`amount` of tokens taken from the `msg.sender`'s balance and locking those for `numSteps` time periods.
The function checks that the address `account` is not the zero address, the `amount` is non-zero and less or equal that `2^96 - 1`, the `numSteps` is non-zero and less than the `MAX_NUM_STEPS`, the `account` has not yet a locked amount, and revert if one of the previous is not satisfied. The `mapLockedBalances` is updated with the new lockedBalances information for the `account` such as the `startTime`, `block.timestamp`, the `endTime, and` the `totalAmount` of the locking. The `total supply` is then calculated and the events and there is a `transferFrom` of the `amount` from the `msg.sender` to this contract. Then the following events are emitted

```
emit Lock(account, amount, block.timestamp, unlockTime);
emit Supply(supplyBefore, supplyAfter);
```

```
    function createLockFor(address account, uint256 amount, uint256
numSteps) external {
        // Check if the account is zero
        if (account == address(0)) {
            revert ZeroAddress();
        }
        // Check if the amount is zero
        if (amount == 0) {
            revert ZeroValue();
        }
        // The locking makes sense for one step or more only
        if (numSteps == 0) {
            revert ZeroValue();
        }
        // Check the maximum number of steps
        if (numSteps > MAX_NUM_STEPS) {
            revert Overflow(numSteps, MAX_NUM_STEPS);
        }
        // Lock time is equal to the number of fixed steps multiply on
a step time
        uint256 unlockTime = block.timestamp + uint256(STEP_TIME) *
numSteps;
        // Max of 2^32 - 1 value, the counter is safe until the year
of 2106
        if (unlockTime > type(uint32).max) {
            revert Overflow(unlockTime, type(uint32).max);
        }
        // After 10 years, the inflation rate is 2% per year. It would
take 220+ years to reach 2^96 - 1 total supply
        if (amount > type(uint96).max) {
```

```
            revert Overflow(amount, type(uint96).max);
        }


        LockedBalance memory lockedBalance =
mapLockedBalances[account];
        // The locked balance must be zero in order to start the lock
        if (lockedBalance.totalAmount > 0) {
            revert LockedValueNotZero(account,
lockedBalance.totalAmount);
        }


        // Store the locked information for the account
        lockedBalance.startTime = uint32(block.timestamp);
        lockedBalance.endTime = uint32(unlockTime);
        lockedBalance.totalAmount = uint96(amount);
        mapLockedBalances[account] = lockedBalance;


        // Calculate total supply
        uint256 supplyBefore = supply;
        uint256 supplyAfter;
        // Cannot overflow because we do not add more tokens than the
OLAS supply
        unchecked {
            supplyAfter = supplyBefore + amount;
            supply = supplyAfter;
        }


        // OLAS is a solmate-based ERC20 token with optimized
transferFrom() that either returns true or reverts
        IERC20(token).transferFrom(msg.sender, address(this), amount);


        emit Lock(account, amount, block.timestamp, unlockTime);
        emit Supply(supplyBefore, supplyAfter);
    }
```

Withdraw

The function `withdraw()` is an external function that allows the `msg.sender`

to realise its matured locked tokens.
Specifically, if the `lockedBalance.startTime` of the `msg.sender` is bigger than zero and the realisable amount is non-zero, that is the balances are still active and not yet fully withdrawn, then either the `lockedBalance.endTime` of the `msg.sender` is bigger than zero meaning that no unrealized amount of the `msg.sender` was revoked or `lockedBalance.endTime` of the `msg.sender` is zero meaning that some tokens of the `msg.sender` was revoked so will be burned with the burn function. In this second case, it is emitted the event
`emit Burn(msg.sender, amountBurn, block.timestamp).`
In both cases, the map `mapLockedBalances` is updating with the new locking information for `msg.sender`. The realisable amount is sent to the `msg.sender` and the total supply is computed. The following events are emitted

```
    emit Withdraw(msg.sender, amount, block.timestamp);
            emit Supply(supplyBefore, supplyAfter);

    function withdraw() external {
        LockedBalance memory lockedBalance =
mapLockedBalances[msg.sender];
        // If the balances are still active and not fully withdrawn,
start time must be greater than zero
        if (lockedBalance.startTime > 0) {
            // Calculate the amount to release
            uint256 amount = _releasableAmount(lockedBalance);
            // Check if at least one locking step has passed
            if (amount == 0) {
                revert LockNotExpired(msg.sender,
lockedBalance.endTime, block.timestamp);
            }


            uint256 supplyBefore = supply;
            uint256 supplyAfter = supplyBefore;
            // End time is greater than zero if withdraw was not fully
completed and `revoke` was not called on the account
            if (lockedBalance.endTime > 0) {
                unchecked {
                    // Update the account locked amount.
                    // Cannot practically overflow since the amount to
release is smaller than the locked amount
                    lockedBalance.transferredAmount += uint96(amount);
                }
                // The balance is fully unlocked. Released amount must
be equal to the locked one
```

```
                if ((lockedBalance.transferredAmount + 1) >
lockedBalance.totalAmount) {
                    mapLockedBalances[msg.sender] = LockedBalance(0,
0, 0, 0);
                } else {
                    mapLockedBalances[msg.sender] = lockedBalance;
                }
            } else {
                // This means revoke has been called on this account
and some tokens must be burned
                uint256 amountBurn =
uint256(lockedBalance.totalAmount);
                // Burn revoked tokens
                if (amountBurn > 0) {
                    IOLAS(token).burn(amountBurn);
                    // Update total supply
                    unchecked {
                        // Amount to burn cannot be bigger than the
supply before the burn
                        supplyAfter = supplyBefore - amountBurn;
                    }
                    emit Burn(msg.sender, amountBurn,
block.timestamp);
                }
                // Set all the data to zero
                mapLockedBalances[msg.sender] = LockedBalance(0, 0, 0,
0);
            }


            // The amount cannot be bigger than the total supply
            unchecked {
                supplyAfter -= amount;
                supply = supplyAfter;
            }


            emit Withdraw(msg.sender, amount, block.timestamp);
            emit Supply(supplyBefore, supplyAfter);

            // OLAS is a solmate-based ERC20 token with optimized
transfer() that either returns true or reverts
            IERC20(token).transfer(msg.sender, amount);
        }
    }
```

Revoke

The function `revoke()` is an external function that allows the owner of the contract to revoke non-matured locked tokens for the list of addressess `accounts`. The function gets the realisable amount of the `accounts` and lets the `lockedBalance.totalAmount` represents the burnable amount. The realisable amount `accounts` of the is sets in `lockedBalance.transferredAmount`. This latter will be transferred to the `accounts` when they call the withdraw function. To flag that non-matured locked tokens are burnable `lockedBalance.endTime` is set to zero. Finally, it is emitted the event `emit Revoke(account, uint256(lockedBalance.totalAmount), block.timestamp)`.

```
    function revoke(address[] memory accounts) external {
        // Check for the ownership
        if (owner != msg.sender) {
            revert OwnerOnly(msg.sender, owner);
        }


        for (uint256 i = 0; i < accounts.length; ++i) {
            address account = accounts[i];
            LockedBalance memory lockedBalance =
mapLockedBalances[account];


            // Get the amount to release
            uint256 amountRelease = _releasableAmount(lockedBalance);
            // Amount locked now represents the burn amount, which can
not become less than zero
            unchecked {
                lockedBalance.totalAmount -= (uint96(amountRelease) +
lockedBalance.transferredAmount);
            }
            // This is the release amount that will be transferred to
the account when they withdraw
            lockedBalance.transferredAmount = uint96(amountRelease);
            // Termination state of the revoke procedure
            lockedBalance.endTime = 0;
            // Update the account data
            mapLockedBalances[account] = lockedBalance;
```

```
            emit Revoke(account, uint256(lockedBalance.totalAmount),
block.timestamp);
        }
    }
```

BalanceOf

The function `balanceOf(address account)` is public and gets the locking balance details of the address `account`. The function is marked as override because the contract call the interface OpenZeppelin - IERC20 .

```
    function balanceOf(address account) public view override returns
(uint256 balance) {
        LockedBalance memory lockedBalance =
mapLockedBalances[account];
        // If the end is equal 0, this balance is either left after
revoke or expired
        if (lockedBalance.endTime == 0) {
            // The maximum balance in this case is the released amount
value
            balance = uint256(lockedBalance.transferredAmount);
        } else {
            // Otherwise the balance is the difference between locked
and released amounts
            balance = uint256(lockedBalance.totalAmount -
lockedBalance.transferredAmount);
        }
    }
```

totalSypply

The function `totalSupply()` is a public function that returns the total token supply. It is marked as override because this calls the interface IERC20.

```
   function totalSupply() public view override returns (uint256) {
        return supply;
    }
```

ReleasableAmount

The function `releasableAmount(address account)` is an external function that returns the realisable token amount of the address `account`

```
    function releasableAmount(address account) external view returns
(uint256 amount) {
        LockedBalance memory lockedBalance =
mapLockedBalances[account];
        amount = _releasableAmount(lockedBalance);
    }
```

_releasableAmount

The function `_releasableAmount(LockedBalance memory lockedBalance)` is a private that computes and returns the releasable amount of the account `lockedBalance` struct.

```
function _releasableAmount(LockedBalance memory lockedBalance) private
view returns (uint256 amount) {
        // If the end is equal 0, this balance is either left after
revoke or expired
        if (lockedBalance.endTime == 0) {
            return lockedBalance.transferredAmount;
        }
        // Number of steps
        uint32 numSteps;
        // Current locked time
        uint32 releasedSteps;
        // Time in the future will be greater than the start time
        unchecked {
            numSteps = (lockedBalance.endTime -
lockedBalance.startTime) / STEP_TIME;
            releasedSteps = (uint32(block.timestamp) -
lockedBalance.startTime) / STEP_TIME;
        }


        // If the number of release steps is greater or equal to the
number of steps, all the available tokens are unlocked
        if ((releasedSteps + 1) > numSteps) {
```

```
            // Return the remainder from the last release since it's
the last one
            unchecked {
                amount = uint256(lockedBalance.totalAmount -
lockedBalance.transferredAmount);
            }
        } else {
            // Calculate the amount to release
            unchecked {
                amount = uint256(lockedBalance.totalAmount *
releasedSteps / numSteps);
                amount -= uint256(lockedBalance.transferredAmount);
            }
        }
    }
```

lockendEnd

The function `lockedEnd(address account)` is an external function. This function returns the scheduled end of the locking time for the user with the address `account`.

```
    function lockedEnd(address account) external view returns (uint256
unlockTime) {
        unlockTime = uint256(mapLockedBalances[account].endTime);
    }
```

supportInterface

The function `supportsInterface(bytes4 interfaceId)` is a public function that gets information about additional interface id that are supported by the contract. It is marked as override because the contract calls the interface `IERC165` that contains the function "`supportsInterface`".

```
    function supportsInterface(bytes4 interfaceId) public view virtual
override returns (bool) {
        return interfaceId == type(IERC20).interfaceId || interfaceId
== type(IVotes).interfaceId ||
            interfaceId == type(IERC165).interfaceId;
    }
```

Transfer

The function `transfer(address to, uint256 amount)` is a public function and it is marked as override because the contract calls the interfaces `IERC20`. This function reverts any transfer of any amount to an address.

```
    function transfer(address to, uint256 amount) external
  virtual override returns (bool) {
        revert NonTransferable(address(this));
    }
```

Approve

The function `approve(address spender, uint256 amount)` is a public function and it is marked as override because the contract calls the interfaces `IERC20`. This function reverts any approval of the spent amount from an address spender

```
    function approve(address spender, uint256 amount) external virtual
override returns (bool) {
        revert NonTransferable(address(this));
    }
```

TransferFrom

The function `transferFrom(address from, address to, uint256 amount)` is a public function and it is marked as override because the contract calls the interfaces `IERC20`. This function reverts transfer from an address to an address of any amount.

```
    function transferFrom(address from, address to, uint256 amount)
external virtual override returns (bool) {
        revert NonTransferable(address(this));
    }
```

Allowance

The function `allowance(address owner, address spender)` is a public function and it is marked as override because the contract calls the interfaces `IERC20`. This function

reverts any change of allowance of the remaining tokens that the spender would be allowed to spend on the owner behalf if approve or transferFrom were called

```
    function allowance(address owner, address spender) external view
virtual override returns (uint256)
    {
        revert NonTransferable(address(this));
    }
```

Contract `Sale`

Contract-defined Interface

The interface ILOCK is an interface for lock functionality.

```
interface ILOCK {
    /// @dev Deposits `amount` tokens for `account` and locks for
`unlockTime` time or number of periods.
    /// @param account Account address.
    /// @param amount Amount to deposit.
    /// @param unlockTime Time or number of time periods when tokens
unlock.
    function createLockFor(address account, uint256 amount, uint256
unlockTime) external;
}
```

The interface IOLAS is an interface for OLAS tokens allowance increases functionality. It includes the function to approve the address `spender` to spend the `amount` of tokens

```
interface IOLAS {
    /// @dev Approves allowance of another account over their tokens.
    /// @param spender Account that tokens are approved for.
    /// @param amount Amount to approve.
    /// @return True if the operation succeeded.
    function approve(address spender, uint256 amount) external returns
(bool);


    /// @dev Gets the amount of tokens owned by `account`.
```

```
    /// @param account Account address.
    /// @return Account balance.
    function balanceOf(address account) external returns (uint256);
}
```

Contract-defined data types

It is created the data type `ClaimableBalance` in the form of a structure for storing the amount of claimable balance, the locking, and the unlocking time. The struct contains a group of elements: `amount` with data type `uint128`, `period` with data type `uint64`.

```
struct ClaimableBalance {
    // Token amount to be locked. Initial OLAS cap is 1 bn tokens, or
1e27.
    // After 10 years, the inflation rate is 2% per year. It would
take 1340+ years to reach 2^128 - 1 total supply
    uint128 amount;
    // Lock time period or number of steps
    // 2^64 - 1 value, which is bigger than the end of time in seconds
while Earth is spinning
    uint64 period;
}
```

Dependencies

The contract `Sale` calls the interface `IErrors`.

Events

The events `CreateVE`, `CreateBU`, `ClaimVE`, `ClaimBU`, and `OwnerUpdated` are declared.

```
contract Sale is IErrors {
    event CreateVE(address indexed account, uint256 amount, uint256
timePeriod);
    event CreateBU(address indexed account, uint256 amount, uint256
numSteps);
    event ClaimVE(address indexed account, uint256 amount, uint256
timePeriod);
```

```
    event ClaimBU(address indexed account, uint256 amount, uint256
numSteps);
    event OwnerUpdated(address indexed owner);
```

Variables

The internal constant `MAX_NUM_STEPS` fixes the maximal number of steps for locking

```
    uint256 internal constant MAX_NUM_STEPS = 10;
```

The internal constant `MINTIME` fixes the minimum locking time in VeOLAS calculated in seconds

```
     uint256 internal constant MINTIME = 365 * 86400;
```

The internal constant `MAXTIME` fixes the maximum locking time in VeOLAS calculated in seconds

```
     uint256 internal constant MAXTIME = 4 * 365 * 86400;
```

The constant `balance` is a public variable that indicates the overall claimable balance

```
    uint256 public supply;
```

The `decimal` is a public variable that set 18 decimals

```
    uint8 public constant decimals = 18;
```

The `olasToken` is a public and immutable variable that indicates the OLAS token address

```
    address public immutable olasToken;
```

The `veToken` is a public and immutable variable that indicates the veOLAS token address

```
    address public immutable veToken;
```

The `buToken` is a public and immutable variable that indicates the buOLAS token address

```
    address public immutable buToken;
```

The `owner` is a public variable that indicates the owner's address

```
    address public owner;
```

Mappings

The public mapping `mapVE` maps the user's `address` to his corresponding `ClaimableBalance` to lock for veOLAS

```
  mapping(address => ClaimableBalance) public mapVE;
```

The public mapping `mapBU` maps the user's `address` to his corresponding `ClaimableBalance` to lock for buOLAS

```
    mapping(address => ClaimableBalance) public mapBU;
```

Constructor

Sets the values for the `address` of the `olasToken`, `veToken`, and `buToken`. The construction calls the following functions to approve `veOLAS` and `buToken` addresses to spend the max `amount` of `type(uint256)` value of `OlasToken` tokens.

```
IOLAS(_olasToken).approve(address(_veToken), type(uint256).max);
IOLAS(_olasToken).approve(address(_buToken),type(uint256).max);


       constructor(address _olasToken, address _veToken, address
_buToken)
    {
        olasToken = _olasToken;
        veToken = _veToken;
        buToken = _buToken;
        owner = msg.sender;
        // Issue allowance for veOLAS and buOLAS. These contracts are
always trusted
        IOLAS(_olasToken).approve(address(_veToken),
type(uint256).max);
        IOLAS(_olasToken).approve(address(_buToken),
type(uint256).max);
    }
```

ChangeOwner

The function `changeOwner(address newOwner)` is an external function that allows the previous owner of the contract to sets a new owner (with a no-zero address) and emit the event

```
emit OwnerUpdated(newOwner);
```

```
    function changeOwner(address newOwner) external {

        if (newOwner == address(0)) {
            revert ZeroAddress();
        }

       if (msg.sender != owner) {
            revert OwnerOnly(msg.sender, owner);
        }

        owner = newOwner;
        emit OwnerUpdated(newOwner);
    }
```

createBalanceFor

The function `createBalancesFor( address[] memory veAccounts, uint256[] memory veAmounts, uint256[] memory veLockTimes, address[] memory buAccounts,uint256[] memory buAmounts, uint256[] memory buNumSteps)` is an external function that creates schedules of locks with `veAmounts` and `veLockTimes` for provided `veAccounts` and creates schedule of locks with `buAmounts` and `buAmounts` for provided `buAccounts`.
Specifically, after checking that the length of the `veAccounts` (respectively `buAccounts`) is the same as `veAmounts` (respectively `buAmounts`) and `veLockTimes` (respectively `buAmounts`) lengths, the function gets the overall balances `veBalance` and `buBalance`. After checking that no account is the zero address, no scheduled amount to lock is zero or bigger than the maximum of the type(uint128), no assigned locking time is bigger than the maximum locking time and less than the minimum (this is just for `veAccounts`), and no locking has already been taking place, the locking information for each address are stored in the `mapVE` (respectively `mapBU`) and the following events are emitted

```
            emit CreateVE(veAccounts[i], veAmounts[i], veLockTimes[i]);
            emit CreateBU(buAccounts[i], buAmounts[i], buNumSteps[i]);
```

Finally, it is checked that the balance of the contract is bigger or equal to the sum of the balances to lock plus the previous balance, and compute the new balance as the previous balance plus the balances to lock.

```
function createBalancesFor(
        address[] memory veAccounts,
        uint256[] memory veAmounts,
        uint256[] memory veLockTimes,
        address[] memory buAccounts,
        uint256[] memory buAmounts,
        uint256[] memory buNumSteps
    ) external {
        // Check for the ownership
        if (owner != msg.sender) {
            revert OwnerOnly(msg.sender, owner);
        }


        // Check that all the corresponding arrays have the same
length
        if (veAccounts.length != veAmounts.length || veAccounts.length
!= veLockTimes.length) {
            revert WrongArrayLength(veAccounts.length,
veAmounts.length);
        }
        if (buAccounts.length != buAmounts.length || buAccounts.length
!= buNumSteps.length) {
            revert WrongArrayLength(buAccounts.length,
buAmounts.length);
        }


        // Get the overall amount balances
        uint256 veBalance;
        uint256 buBalance;


        // Create lock-ready structures for veOLAS
        for (uint256 i = 0; i < veAccounts.length; ++i) {
            // Check for the zero addresses
            if (veAccounts[i] == address(0)) {
```

```
                revert ZeroAddress();
            }
            // Check for other zero values
            if (veAmounts[i] == 0) {
                revert ZeroValue();
            }
            // Check for the amount bounds
            if (veAmounts[i] > type(uint128).max) {
                revert Overflow(veAmounts[i], type(uint128).max);
            }
            // Check the end of a lock time
            if (veLockTimes[i] < MINTIME) {
                revert UnlockTimeIncorrect(veAccounts[i], MINTIME,
veLockTimes[i]);
            }
            if (veLockTimes[i] > MAXTIME) {
                revert MaxUnlockTimeReached(veAccounts[i], MAXTIME,
veLockTimes[i]);
            }
            // Check if the lock has already been placed
            ClaimableBalance memory lockedBalance =
mapVE[veAccounts[i]];
            if (lockedBalance.amount > 0) {
                revert NonZeroValue();
            }


            // Update allowance, push values to the dedicated locking
slot
            veBalance += veAmounts[i];
            lockedBalance.amount = uint128(veAmounts[i]);
            lockedBalance.period = uint64(veLockTimes[i]);
            mapVE[veAccounts[i]] = lockedBalance;


            emit CreateVE(veAccounts[i], veAmounts[i],
veLockTimes[i]);
        }


        // Create lock-ready structures for buOLAS
        for (uint256 i = 0; i < buAccounts.length; ++i) {
            // Check for the zero addresses
            if (buAccounts[i] == address(0)) {
                revert ZeroAddress();
```

```
            }
            // Check for other zero values
            if (buAmounts[i] == 0 || buNumSteps[i] == 0) {
                revert ZeroValue();
            }
            // Check for the amount bounds
            if (buAmounts[i] > type(uint128).max) {
                revert Overflow(buAmounts[i], type(uint128).max);
            }
            // Check for the number of lock steps
            if (buNumSteps[i] > MAX_NUM_STEPS) {
                revert Overflow(buNumSteps[i], MAX_NUM_STEPS);
            }
            // Check if the lock has already been placed
            ClaimableBalance memory lockedBalance =
mapBU[buAccounts[i]];
            if (lockedBalance.amount > 0) {
                revert NonZeroValue();
            }


            // Update allowance, push values to the dedicated locking
slot
            buBalance += buAmounts[i];
            lockedBalance.amount = uint128(buAmounts[i]);
            lockedBalance.period = uint64(buNumSteps[i]);
            mapBU[buAccounts[i]] = lockedBalance;


            emit CreateBU(buAccounts[i], buAmounts[i], buNumSteps[i]);
        }


        // Own balance cannot be smaller than the sum of balances for
all the accounts plus the previous balance
        uint256 curBalance =
IOLAS(olasToken).balanceOf(address(this));
        uint256 balanceAfter = balance + buBalance + veBalance;
        if (curBalance < balanceAfter) {
            revert InsufficientAllowance(balanceAfter, curBalance);
        }
        balance = balanceAfter;
    }
```

Claim

The function `claim()` is an external function that allows the `msg.sender` to claim token lock into veOLAS and buOLAS contracts. Specifically, the function gets the `msg.sender's ClaimableBalance` to lock for veOLAS (respectively buOLAS) and when the claimable amount of the locked balance is non-zero, it calls the createLockFor function to create the veOLAS (respectively buOLAS) locking for `msg.sender` with the amount equal to the claimable amount of the `msg.sender` and the locking time equal to the prescribed locking period of the `msg.sender`. Then the `ClaimableBalance` information of the `msg.sender` is set to zero in the `mapVE` (respectively `mapBU`) and the following events are emitted

```
emit ClaimVE(msg.sender, uint256(lockedBalance.amount),
uint256(lockedBalance.period));

emit ClaimBU(msg.sender, uint256(lockedBalance.amount),
uint256(lockedBalance.period));
```

The function checks if there is still a claimable balance and updates the balance by subtracting the claimable one.

```
      function claim() external {
        uint256 balanceClaim;
        // Get the balance, lock time and call the veOLAS locking
function
        ClaimableBalance memory lockedBalance = mapVE[msg.sender];
        if (lockedBalance.amount > 0) {
            // We need to update the balance tracker
            balanceClaim = uint256(lockedBalance.amount);
            ILOCK(veToken).createLockFor(msg.sender,
uint256(lockedBalance.amount), uint256(lockedBalance.period));
            mapVE[msg.sender] = ClaimableBalance(0, 0);
            emit ClaimVE(msg.sender, uint256(lockedBalance.amount),
uint256(lockedBalance.period));
        }


        lockedBalance = mapBU[msg.sender];
        if (lockedBalance.amount > 0) {
            balanceClaim += uint256(lockedBalance.amount);
            ILOCK(buToken).createLockFor(msg.sender,
uint256(lockedBalance.amount), uint256(lockedBalance.period));
            mapBU[msg.sender] = ClaimableBalance(0, 0);
```

```
            emit ClaimBU(msg.sender, uint256(lockedBalance.amount),
uint256(lockedBalance.period));
        }


        // Check if anything was claimed
        if (balanceClaim == 0) {
            revert ZeroValue();
        }


        // The overall balance can not be smaller than the claimable
balance, since createBalancesFor would revert before
        unchecked {
            balance -= balanceClaim;
        }
    }
```

ClaimableBalances

The function `claimableBalances(address account)` is an external function that gets the veOLAS and buOLAS claimable balances of the address `account`.

```
    function claimableBalances(address account) external view returns
(uint256 veBalance, uint256 buBalance) {
        veBalance = uint256(mapVE[account].amount);
        buBalance = uint256(mapBU[account].amount);
    }
}
```

Contract `OLAS`

The contract imports the Solmate ERC20 token library.

Contract-defined error

`error ManagerOnly(address sender, address manager);`
Only `manager` has a privilege to take some actions. So `ManagerOnly(address`

`sender, address manager)` is emitted when an `address sender` different from the `address manager` tries an action reserved to the manager.

`error ZeroAddress();`
Some actions require that the address can't be the zero address.

Dependencies

The contract is a Solmate-based `ERC20` token.

Events

The events `MinterUpdated` and `OwnerUpdated` are declared.

```
event MinterUpdated(address indexed minter);
event OwnerUpdated(address indexed owner);
```

Variables

The public constant `oneYear` sets one year calculated one year in days

```
uint256 public constant oneYear = 1 days * 365;
```

The public constant `tenYearSupplyCap` sets the total supply cap for the first ten years at 1 bn (including 18 decimal, as for ERC20 standard, this gives a total of 10^27 units)

```
uint256 public constant tenYearSupplyCap = 1_000_000_000e18;
```

The `maxMintCapFraction` is a public constant thet fix the maximum of annual inflation after ten years

```
uint256 public constant maxMintCapFraction = 2;
```

The `timeLaunch` is a public and immutable variable that indicates the initial timestamp of the token deployment

```
uint256 public immutable timeLaunch;
```

The `owner` is a public variable that indicates the owner's address

```
    address public owner;
```

The `minter` is a public variable that indicates the minter's address

```
    address public minter;
```

Constructor

This contract is ownable with `owner =minter = msg.sender` and the `timeLaunch` is the timestamp of the token deployment

```
    constructor() ERC20("Autonolas", "OLAS", 18) {
        owner = msg.sender;
        minter = msg.sender;
        timeLaunch = block.timestamp;
    }
```

ChangeOwner

The function `changeOwner(address newOwner)` is an external function that allows the previous owner of the contract to sets a new owner (with a no-zero address) and emit the event

```
 emit OwnerUpdated(newOwner);
```

```
    function changeOwner(address newOwner) external {

        if (newOwner == address(0)) {
            revert ZeroAddress();
        }

       if (msg.sender != owner) {
            revert OwnerOnly(msg.sender, owner);
        }

        owner = newOwner;
        emit OwnerUpdated(newOwner);
    }
```

ChangeMinter

The function `changeMinter(address newMinter)` is an external function that allows the owner of the contract to sets a new minter (with a no-zero address) and emit the event `emit MinterUpdated(newMinter)`

```
    function changeMinter(address newMinter) external {
        if (msg.sender != owner) {
            revert ManagerOnly(msg.sender, owner);
        }


        if (newMinter == address(0)) {
            revert ZeroAddress();
        }


        minter = newMinter;
        emit MinterUpdated(newMinter);
    }
```

Mint

The function `mint(address account, uint256 amount)` is an external function that allows the `minter` to mint new tokens respecting the inflation constraints.

```
    function mint(address account, uint256 amount) external {
        // Access control
        if (msg.sender != minter) {
            revert ManagerOnly(msg.sender, minter);
        }


        // Check the inflation schedule and mint
        if (inflationControl(amount)) {
            _mint(account, amount);
        }
    }
```

InflationControl

The function `inflationControl(uint256 amount)` is a public function that checks if the `amount` is less than `inflationRemainder`.

```
    function inflationControl(uint256 amount) public view returns
(bool) {
        uint256 remainder = inflationRemainder();
        return (amount <= remainder);
    }
```

InflationControl

The function `inflationRemainder()` is a public function that checks the remainder of OLAS that can be minted with respect to the inflation schedule.

```
    function inflationRemainder() public view returns (uint256
remainder) {
        uint256 _totalSupply = totalSupply;
        // Current year
        uint256 numYears = (block.timestamp - timeLaunch) / oneYear;
        // Calculate maximum mint amount to date
        uint256 supplyCap = tenYearSupplyCap;
        // After 10 years, adjust supplyCap according to the yearly
inflation % set in maxMintCapFraction
        if (numYears > 9) {
            // Number of years after ten years have passed (including
ongoing ones)
            numYears -= 9;
            for (uint256 i = 0; i < numYears; ++i) {
                supplyCap += (supplyCap * maxMintCapFraction) / 100;
            }
        }
        // Check for the requested mint overflow
        remainder = supplyCap - _totalSupply;
    }
```

Burn

The function `burn()` is a public function that calls the Solmate function `_burn(address from, uint256 amount)` that burns the `amount` of tokens from the `msg.sender` and then subtracts the `amount` from the total supply.

DecreasesAllowance

The function`decreaseAllowance(address spender, uint256 amount)` is an external function that allows reducing the quantity `amount` from the allowance previously given by the `msg.sender` to the address `spender`. The following event is emitted

```
emit Approval(msg.sender, spender, spenderAllowance);
```

and the function returns if the operation succeeded.

```
    function decreaseAllowance(address spender, uint256 amount)
external returns (bool) {
        uint256 spenderAllowance = allowance[msg.sender][spender];


        if (spenderAllowance != type(uint256).max) {
            spenderAllowance -= amount;
            allowance[msg.sender][spender] = spenderAllowance;
            emit Approval(msg.sender, spender, spenderAllowance);
        }


        return true;
    }
```

IncreasesAllowance

The function `increasesAllowance(address spender, uint256 amount)` is an external function that allows increasing the quantity `amount` form the allowance previously given by the `msg.sender` to the address `spender`. The following event is emitted

```
emit Approval(msg.sender, spender, spenderAllowance);
```

and the function returns if the operation succeeded.

```
    function increaseAllowance(address spender, uint256 amount)
external returns (bool) {
        uint256 spenderAllowance = allowance[msg.sender][spender];


        spenderAllowance += amount;
        allowance[msg.sender][spender] = spenderAllowance;
        emit Approval(msg.sender, spender, spenderAllowance);


        return true;
    }
}
```