DRB Token Contract – Full Verified Source Code

Token Name: Debt Relief Bot ($DRB)
Contract Address: 0x3ec2156d4c0a9cbdab4a016633b7bcf6a8d68ea2
Base Chain

This document contains the complete, verified source code for the $DRB ERC-20 token contract (ClankerToken template – 33 files total).
Fully verified on Basescan:
https://basescan.org/address/0x3ec2156d4c0a9cbdab4a016633b7bcf6a8d68ea2#code

Key Security Features:
- Fixed supply of 100,000,000,000 tokens
- No public mint function
- No upgradeability (immutable code)
- No owner/admin privileges after deployment
- No hidden transfer fees or taxes

Prepared for exchange listing review.
Date: March 2026

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.25;
import {ERC20} from "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import {IERC165} from "@openzeppelin/contracts/interfaces/IERC165.sol";
import {ERC20Votes} from
"@openzeppelin/contracts/token/ERC20/extensions/ERC20Votes.sol";
import {ERC20Permit} from
"@openzeppelin/contracts/token/ERC20/extensions/ERC20Permit.sol";
import {ERC20Burnable} from
"@openzeppelin/contracts/token/ERC20/extensions/ERC20Burnable.sol";
import {Nonces} from "@openzeppelin/contracts/utils/Nonces.sol";
import { IERC7802 } from "@contracts-bedrock/interfaces/L2/IERC7802.sol";
import { Predeploys } from "@contracts-bedrock/src/libraries/Predeploys.sol";
import { Unauthorized } from "@contracts-bedrock/src/libraries/errors/CommonErrors.sol";
contract ClankerToken is ERC20, ERC20Permit, ERC20Votes, ERC20Burnable, IERC7802 {
    error NotDeployer();
    string private _name;
    string private _symbol;
    uint8 private immutable _decimals;
```

```
address private _deployer;
uint256 private _fid;
string private _image;
string private _castHash;
constructor(
    string memory name_,
    string memory symbol_,
    uint256 maxSupply_,
    address deployer_,
    uint256 fid_,
    string memory image_,
    string memory castHash_
) ERC20(name_, symbol_) ERC20Permit(name_) {
    _deployer = deployer_;
    _fid = fid_;
    _image = image_;
    _castHash = castHash_;
    _mint(msg.sender, maxSupply_);
}
function updateImage(string memory image_) public {
    if (msg.sender != _deployer) {
        revert NotDeployer();
    }
    _image = image_;
}
function _update(
    address from,
    address to,
    uint256 value
) internal override(ERC20, ERC20Votes) {
    super._update(from, to, value);
}
function nonces(
    address owner
) public view virtual override(ERC20Permit, Nonces) returns (uint256) {
    return super.nonces(owner);
}
function fid() public view returns (uint256) {
    return _fid;
}
function deployer() public view returns (address) {
    return _deployer;
}
function image() public view returns (string memory) {
```

```
        return _image;
    }
    function castHash() public view returns (string memory) {
        return _castHash;
    }
    function crosschainMint(address _to, uint256 _amount) external {
        // Only the `SuperchainTokenBridge` has permissions to mint tokens during crosschain
transfers.
        if (msg.sender != Predeploys.SUPERCHAIN_TOKEN_BRIDGE) revert Unauthorized();

        // Mint tokens to the `_to` account's balance.
        _mint(_to, _amount);
        // Emit the CrosschainMint event included on IERC7802 for tracking token mints associated
with cross chain transfers.
        emit CrosschainMint(_to, _amount, msg.sender);
    }
    function crosschainBurn(address _from, uint256 _amount) external {
        // Only the `SuperchainTokenBridge` has permissions to burn tokens during crosschain
transfers.
        if (msg.sender != Predeploys.SUPERCHAIN_TOKEN_BRIDGE) revert Unauthorized();
        // Burn the tokens from the `_from` account's balance.
        _burn(_from, _amount);
        // Emit the CrosschainBurn event included on IERC7802 for tracking token burns
associated with cross chain transfers.
        emit CrosschainBurn(_from, _amount, msg.sender);
    }
    function supportsInterface(bytes4 _interfaceId) public pure returns (bool) {
        return _interfaceId == type(IERC7802).interfaceId || _interfaceId ==
type(IERC20).interfaceId
            || _interfaceId == type(IERC165).interfaceId;
    }
}


// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts (last updated v5.1.0) (token/ERC20/ERC20.sol)
pragma solidity ^0.8.20;
import {IERC20} from "./IERC20.sol";
import {IERC20Metadata} from "./extensions/IERC20Metadata.sol";
import {Context} from "../../utils/Context.sol";
import {IERC20Errors} from "../../interfaces/draft-IERC6093.sol";
/**
 * @dev Implementation of the {IERC20} interface.
 *
```

```
 * This implementation is agnostic to the way tokens are created. This means
 * that a supply mechanism has to be added in a derived contract using {_mint}.
 *
 * TIP: For a detailed writeup see our guide
 * https://forum.openzeppelin.com/t/how-to-implement-erc20-supply-mechanisms/226[How
 * to implement supply mechanisms].
 *
 * The default value of {decimals} is 18. To change this, you should override
 * this function so it returns a different value.
 *
 * We have followed general OpenZeppelin Contracts guidelines: functions revert
 * instead returning `false` on failure. This behavior is nonetheless
 * conventional and does not conflict with the expectations of ERC-20
 * applications.
 */
abstract contract ERC20 is Context, IERC20, IERC20Metadata, IERC20Errors {
    mapping(address account => uint256) private _balances;
    mapping(address account => mapping(address spender => uint256)) private _allowances;
    uint256 private _totalSupply;
    string private _name;
    string private _symbol;
    /**
     * @dev Sets the values for {name} and {symbol}.
     *
     * All two of these values are immutable: they can only be set once during
     * construction.
     */
    constructor(string memory name_, string memory symbol_) {
        _name = name_;
        _symbol = symbol_;
    }
    /**
     * @dev Returns the name of the token.
     */
    function name() public view virtual returns (string memory) {
        return _name;
    }
    /**
     * @dev Returns the symbol of the token, usually a shorter version of the
     * name.
     */
    function symbol() public view virtual returns (string memory) {
        return _symbol;
    }
```

```
/**
 * @dev Returns the number of decimals used to get its user representation.
 * For example, if `decimals` equals `2`, a balance of `505` tokens should
 * be displayed to a user as `5.05` (`505 / 10 ** 2`).
 *
 * Tokens usually opt for a value of 18, imitating the relationship between
 * Ether and Wei. This is the default value returned by this function, unless
 * it's overridden.
 *
 * NOTE: This information is only used for _display_ purposes: it in
 * no way affects any of the arithmetic of the contract, including
 * {IERC20-balanceOf} and {IERC20-transfer}.
 */
function decimals() public view virtual returns (uint8) {
    return 18;
}
/**
 * @dev See {IERC20-totalSupply}.
 */
function totalSupply() public view virtual returns (uint256) {
    return _totalSupply;
}
/**
 * @dev See {IERC20-balanceOf}.
 */
function balanceOf(address account) public view virtual returns (uint256) {
    return _balances[account];
}
/**
 * @dev See {IERC20-transfer}.
 *
 * Requirements:
 *
 * - `to` cannot be the zero address.
 * - the caller must have a balance of at least `value`.
 */
function transfer(address to, uint256 value) public virtual returns (bool) {
    address owner = _msgSender();
    _transfer(owner, to, value);
    return true;
}
/**
 * @dev See {IERC20-allowance}.
 */
```

```
    function allowance(address owner, address spender) public view virtual returns (uint256) {
        return _allowances[owner][spender];
    }
    /**
     * @dev See {IERC20-approve}.
     *
     * NOTE: If `value` is the maximum `uint256`, the allowance is not updated on
     * `transferFrom`. This is semantically equivalent to an infinite approval.
     *
     * Requirements:
     *
     * - `spender` cannot be the zero address.
     */
    function approve(address spender, uint256 value) public virtual returns (bool) {
        address owner = _msgSender();
        _approve(owner, spender, value);
        return true;
    }
    /**
     * @dev See {IERC20-transferFrom}.
     *
     * Skips emitting an {Approval} event indicating an allowance update. This is not
     * required by the ERC. See
{xref-ERC20-_approve-address-address-uint256-bool-}[_approve].
     *
     * NOTE: Does not update the allowance if the current allowance
     * is the maximum `uint256`.
     *
     * Requirements:
     *
     * - `from` and `to` cannot be the zero address.
     * - `from` must have a balance of at least `value`.
     * - the caller must have allowance for ``from``'s tokens of at least
     * `value`.
     */
    function transferFrom(address from, address to, uint256 value) public virtual returns (bool) {
        address spender = _msgSender();
        _spendAllowance(from, spender, value);
        _transfer(from, to, value);
        return true;
    }
    /**
     * @dev Moves a `value` amount of tokens from `from` to `to`.
     *
```

```
     * This internal function is equivalent to {transfer}, and can be used to
     * e.g. implement automatic token fees, slashing mechanisms, etc.
     *
     * Emits a {Transfer} event.
     *
     * NOTE: This function is not virtual, {_update} should be overridden instead.
     */
    function _transfer(address from, address to, uint256 value) internal {
        if (from == address(0)) {
            revert ERC20InvalidSender(address(0));
        }
        if (to == address(0)) {
            revert ERC20InvalidReceiver(address(0));
        }
        _update(from, to, value);
    }
    /**
     * @dev Transfers a `value` amount of tokens from `from` to `to`, or alternatively mints (or
burns) if `from`
     * (or `to`) is the zero address. All customizations to transfers, mints, and burns should be
done by overriding
     * this function.
     *
     * Emits a {Transfer} event.
     */
    function _update(address from, address to, uint256 value) internal virtual {
        if (from == address(0)) {
            // Overflow check required: The rest of the code assumes that totalSupply never
overflows
            _totalSupply += value;
        } else {
            uint256 fromBalance = _balances[from];
            if (fromBalance < value) {
                revert ERC20InsufficientBalance(from, fromBalance, value);
            }
            unchecked {
                // Overflow not possible: value <= fromBalance <= totalSupply.
                _balances[from] = fromBalance - value;
            }
        }
        if (to == address(0)) {
            unchecked {
                // Overflow not possible: value <= totalSupply or value <= fromBalance <= totalSupply.
                _totalSupply -= value;
```

```
            }
        } else {
            unchecked {
                // Overflow not possible: balance + value is at most totalSupply, which we know fits into a uint256.
                _balances[to] += value;
            }
        }
        emit Transfer(from, to, value);
    }
    /**
     * @dev Creates a `value` amount of tokens and assigns them to `account`, by transferring it from address(0).
     * Relies on the `_update` mechanism
     *
     * Emits a {Transfer} event with `from` set to the zero address.
     *
     * NOTE: This function is not virtual, {_update} should be overridden instead.
     */
    function _mint(address account, uint256 value) internal {
        if (account == address(0)) {
            revert ERC20InvalidReceiver(address(0));
        }
        _update(address(0), account, value);
    }
    /**
     * @dev Destroys a `value` amount of tokens from `account`, lowering the total supply.
     * Relies on the `_update` mechanism.
     *
     * Emits a {Transfer} event with `to` set to the zero address.
     *
     * NOTE: This function is not virtual, {_update} should be overridden instead
     */
    function _burn(address account, uint256 value) internal {
        if (account == address(0)) {
            revert ERC20InvalidSender(address(0));
        }
        _update(account, address(0), value);
    }
    /**
     * @dev Sets `value` as the allowance of `spender` over the `owner` s tokens.
     *
     * This internal function is equivalent to `approve`, and can be used to
     * e.g. set automatic allowances for certain subsystems, etc.
```

```
     *
     * Emits an {Approval} event.
     *
     * Requirements:
     *
     * - `owner` cannot be the zero address.
     * - `spender` cannot be the zero address.
     *
     * Overrides to this logic should be done to the variant with an additional `bool emitEvent`
argument.
     */
    function _approve(address owner, address spender, uint256 value) internal {
        _approve(owner, spender, value, true);
    }
    /**
     * @dev Variant of {_approve} with an optional flag to enable or disable the {Approval} event.
     *
     * By default (when calling {_approve}) the flag is set to true. On the other hand, approval
changes made by
     * `_spendAllowance` during the `transferFrom` operation set the flag to false. This saves gas
by not emitting any
     * `Approval` event during `transferFrom` operations.
     *
     * Anyone who wishes to continue emitting `Approval` events on the`transferFrom` operation
can force the flag to
     * true using the following override:
     *
     * ```solidity
     * function _approve(address owner, address spender, uint256 value, bool) internal virtual
override {
     *     super._approve(owner, spender, value, true);
     * }
     * ```
     *
     * Requirements are the same as {_approve}.
     */
    function _approve(address owner, address spender, uint256 value, bool emitEvent) internal
virtual {
        if (owner == address(0)) {
            revert ERC20InvalidApprover(address(0));
        }
        if (spender == address(0)) {
            revert ERC20InvalidSpender(address(0));
        }
```

```
        _allowances[owner][spender] = value;
        if (emitEvent) {
            emit Approval(owner, spender, value);
        }
    }
    /**
     * @dev Updates `owner` s allowance for `spender` based on spent `value`.
     *
     * Does not update the allowance value in case of infinite allowance.
     * Revert if not enough allowance is available.
     *
     * Does not emit an {Approval} event.
     */
    function _spendAllowance(address owner, address spender, uint256 value) internal virtual {
        uint256 currentAllowance = allowance(owner, spender);
        if (currentAllowance != type(uint256).max) {
            if (currentAllowance < value) {
                revert ERC20InsufficientAllowance(spender, currentAllowance, value);
            }
            unchecked {
                _approve(owner, spender, currentAllowance - value, false);
            }
        }
    }
}


// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts (last updated v5.1.0) (token/ERC20/IERC20.sol)
pragma solidity ^0.8.20;
/**
 * @dev Interface of the ERC-20 standard as defined in the ERC.
 */
interface IERC20 {
    /**
     * @dev Emitted when `value` tokens are moved from one account (`from`) to
     * another (`to`).
     *
     * Note that `value` may be zero.
     */
    event Transfer(address indexed from, address indexed to, uint256 value);
    /**
     * @dev Emitted when the allowance of a `spender` for an `owner` is set by
     * a call to {approve}. `value` is the new allowance.
```

```
 */
event Approval(address indexed owner, address indexed spender, uint256 value);
/**
 * @dev Returns the value of tokens in existence.
 */
function totalSupply() external view returns (uint256);
/**
 * @dev Returns the value of tokens owned by `account`.
 */
function balanceOf(address account) external view returns (uint256);
/**
 * @dev Moves a `value` amount of tokens from the caller's account to `to`.
 *
 * Returns a boolean value indicating whether the operation succeeded.
 *
 * Emits a {Transfer} event.
 */
function transfer(address to, uint256 value) external returns (bool);
/**
 * @dev Returns the remaining number of tokens that `spender` will be
 * allowed to spend on behalf of `owner` through {transferFrom}. This is
 * zero by default.
 *
 * This value changes when {approve} or {transferFrom} are called.
 */
function allowance(address owner, address spender) external view returns (uint256);
/**
 * @dev Sets a `value` amount of tokens as the allowance of `spender` over the
 * caller's tokens.
 *
 * Returns a boolean value indicating whether the operation succeeded.
 *
 * IMPORTANT: Beware that changing an allowance with this method brings the risk
 * that someone may use both the old and the new allowance by unfortunate
 * transaction ordering. One possible solution to mitigate this race
 * condition is to first reduce the spender's allowance to 0 and set the
 * desired value afterwards:
 * https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
 *
 * Emits an {Approval} event.
 */
function approve(address spender, uint256 value) external returns (bool);
/**
 * @dev Moves a `value` amount of tokens from `from` to `to` using the
```

```
     * allowance mechanism. `value` is then deducted from the caller's
     * allowance.
     *
     * Returns a boolean value indicating whether the operation succeeded.
     *
     * Emits a {Transfer} event.
     */
    function transferFrom(address from, address to, uint256 value) external returns (bool);
}


// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts (last updated v5.0.0) (interfaces/IERC165.sol)
pragma solidity ^0.8.20;
import {IERC165} from "../utils/introspection/IERC165.sol";


// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts (last updated v5.1.0) (token/ERC20/extensions/ERC20Votes.sol)
pragma solidity ^0.8.20;
import {ERC20} from "../ERC20.sol";
import {Votes} from "../../../governance/utils/Votes.sol";
import {Checkpoints} from "../../../utils/structs/Checkpoints.sol";
/**
 * @dev Extension of ERC-20 to support Compound-like voting and delegation. This version is
more generic than Compound's,
 * and supports token supply up to 2^208^ - 1, while COMP is limited to 2^96^ - 1.
 *
 * NOTE: This contract does not provide interface compatibility with Compound's COMP token.
 *
 * This extension keeps a history (checkpoints) of each account's vote power. Vote power can
be delegated either
 * by calling the {Votes-delegate} function directly, or by providing a signature to be used with
{Votes-delegateBySig}. Voting
 * power can be queried through the public accessors {Votes-getVotes} and
{Votes-getPastVotes}.
 *
 * By default, token balance does not account for voting power. This makes transfers cheaper.
The downside is that it
 * requires users to delegate to themselves in order to activate checkpoints and have their
voting power tracked.
 */
abstract contract ERC20Votes is ERC20, Votes {
    /**
```

```
     * @dev Total supply cap has been exceeded, introducing a risk of votes overflowing.
     */
    error ERC20ExceededSafeSupply(uint256 increasedSupply, uint256 cap);
    /**
     * @dev Maximum token supply. Defaults to `type(uint208).max` (2^208^ - 1).
     *
     * This maximum is enforced in {_update}. It limits the total supply of the token, which is otherwise a uint256,
     * so that checkpoints can be stored in the Trace208 structure used by {Votes}. Increasing this value will not
     * remove the underlying limitation, and will cause {_update} to fail because of a math overflow in
     * {Votes-_transferVotingUnits}. An override could be used to further restrict the total supply (to a lower value) if
     * additional logic requires it. When resolving override conflicts on this function, the minimum should be
     * returned.
     */
    function _maxSupply() internal view virtual returns (uint256) {
        return type(uint208).max;
    }
    /**
     * @dev Move voting power when tokens are transferred.
     *
     * Emits a {IVotes-DelegateVotesChanged} event.
     */
    function _update(address from, address to, uint256 value) internal virtual override {
        super._update(from, to, value);
        if (from == address(0)) {
            uint256 supply = totalSupply();
            uint256 cap = _maxSupply();
            if (supply > cap) {
                revert ERC20ExceededSafeSupply(supply, cap);
            }
        }
        _transferVotingUnits(from, to, value);
    }
    /**
     * @dev Returns the voting units of an `account`.
     *
     * WARNING: Overriding this function may compromise the internal vote accounting.
     * `ERC20Votes` assumes tokens map to voting units 1:1 and this is not easy to change.
     */
    function _getVotingUnits(address account) internal view virtual override returns (uint256) {
```

```
        return balanceOf(account);
    }
    /**
     * @dev Get number of checkpoints for `account`.
     */
    function numCheckpoints(address account) public view virtual returns (uint32) {
        return _numCheckpoints(account);
    }
    /**
     * @dev Get the `pos`-th checkpoint for `account`.
     */
    function checkpoints(address account, uint32 pos) public view virtual returns
(Checkpoints.Checkpoint208 memory) {
        return _checkpoints(account, pos);
    }
}


// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts (last updated v5.1.0) (token/ERC20/extensions/ERC20Permit.sol)
pragma solidity ^0.8.20;
import {IERC20Permit} from "./IERC20Permit.sol";
import {ERC20} from "../ERC20.sol";
import {ECDSA} from "../../../utils/cryptography/ECDSA.sol";
import {EIP712} from "../../../utils/cryptography/EIP712.sol";
import {Nonces} from "../../../utils/Nonces.sol";
/**
 * @dev Implementation of the ERC-20 Permit extension allowing approvals to be made via
signatures, as defined in
 * https://eips.ethereum.org/EIPS/eip-2612[ERC-2612].
 *
 * Adds the {permit} method, which can be used to change an account's ERC-20 allowance (see
{IERC20-allowance}) by
 * presenting a message signed by the account. By not relying on `{IERC20-approve}`, the token
holder account doesn't
 * need to send a transaction, and thus is not required to hold Ether at all.
 */
abstract contract ERC20Permit is ERC20, IERC20Permit, EIP712, Nonces {
    bytes32 private constant PERMIT_TYPEHASH =
        keccak256("Permit(address owner,address spender,uint256 value,uint256 nonce,uint256
deadline)");
    /**
     * @dev Permit deadline has expired.
     */
```

```
    error ERC2612ExpiredSignature(uint256 deadline);
    /**
     * @dev Mismatched signature.
     */
    error ERC2612InvalidSigner(address signer, address owner);
    /**
     * @dev Initializes the {EIP712} domain separator using the `name` parameter, and setting
`version` to `"1"`.
     *
     * It's a good idea to use the same `name` that is defined as the ERC-20 token name.
     */
    constructor(string memory name) EIP712(name, "1") {}
    /**
     * @inheritdoc IERC20Permit
     */
    function permit(
        address owner,
        address spender,
        uint256 value,
        uint256 deadline,
        uint8 v,
        bytes32 r,
        bytes32 s
    ) public virtual {
        if (block.timestamp > deadline) {
            revert ERC2612ExpiredSignature(deadline);
        }
        bytes32 structHash = keccak256(abi.encode(PERMIT_TYPEHASH, owner, spender, value,
_useNonce(owner), deadline));
        bytes32 hash = _hashTypedDataV4(structHash);
        address signer = ECDSA.recover(hash, v, r, s);
        if (signer != owner) {
            revert ERC2612InvalidSigner(signer, owner);
        }
        _approve(owner, spender, value);
    }
    /**
     * @inheritdoc IERC20Permit
     */
    function nonces(address owner) public view virtual override(IERC20Permit, Nonces) returns
(uint256) {
        return super.nonces(owner);
    }
    /**
```

```
     * @inheritdoc IERC20Permit
     */
    // solhint-disable-next-line func-name-mixedcase
    function DOMAIN_SEPARATOR() external view virtual returns (bytes32) {
        return _domainSeparatorV4();
    }
}


// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts (last updated v5.0.0) (token/ERC20/extensions/ERC20Burnable.sol)
pragma solidity ^0.8.20;
import {ERC20} from "../ERC20.sol";
import {Context} from "../../../utils/Context.sol";
/**
 * @dev Extension of {ERC20} that allows token holders to destroy both their own
 * tokens and those that they have an allowance for, in a way that can be
 * recognized off-chain (via event analysis).
 */
abstract contract ERC20Burnable is Context, ERC20 {
    /**
     * @dev Destroys a `value` amount of tokens from the caller.
     *
     * See {ERC20-_burn}.
     */
    function burn(uint256 value) public virtual {
        _burn(_msgSender(), value);
    }
    /**
     * @dev Destroys a `value` amount of tokens from `account`, deducting from
     * the caller's allowance.
     *
     * See {ERC20-_burn} and {ERC20-allowance}.
     *
     * Requirements:
     *
     * - the caller must have allowance for ``accounts``'s tokens of at least
     * `value`.
     */
    function burnFrom(address account, uint256 value) public virtual {
        _spendAllowance(account, _msgSender(), value);
        _burn(account, value);
    }
}
```

```
// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts (last updated v5.0.0) (utils/Nonces.sol)
pragma solidity ^0.8.20;
/**
 * @dev Provides tracking nonces for addresses. Nonces will only increment.
 */
abstract contract Nonces {
    /**
     * @dev The nonce used for an `account` is not the expected current nonce.
     */
    error InvalidAccountNonce(address account, uint256 currentNonce);
    mapping(address account => uint256) private _nonces;
    /**
     * @dev Returns the next unused nonce for an address.
     */
    function nonces(address owner) public view virtual returns (uint256) {
        return _nonces[owner];
    }
    /**
     * @dev Consumes a nonce.
     *
     * Returns the current value and increments nonce.
     */
    function _useNonce(address owner) internal virtual returns (uint256) {
        // For each account, the nonce has an initial value of 0, can only be incremented by one,
and cannot be
        // decremented or reset. This guarantees that the nonce never overflows.
        unchecked {
            // It is important to do x++ and not ++x here.
            return _nonces[owner]++;
        }
    }
    /**
     * @dev Same as {_useNonce} but checking that `nonce` is the next valid for `owner`.
     */
    function _useCheckedNonce(address owner, uint256 nonce) internal virtual {
        uint256 current = _useNonce(owner);
        if (nonce != current) {
            revert InvalidAccountNonce(owner, current);
        }
    }
}
```

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
import { IERC165 } from "@openzeppelin/contracts/interfaces/IERC165.sol";
/// @title IERC7802
/// @notice Defines the interface for crosschain ERC20 transfers.
interface IERC7802 is IERC165 {
    /// @notice Emitted when a crosschain transfer mints tokens.
    /// @param to       Address of the account tokens are being minted for.
    /// @param amount   Amount of tokens minted.
    /// @param sender   Address of the account that finilized the crosschain transfer.
    event CrosschainMint(address indexed to, uint256 amount, address indexed sender);
    /// @notice Emitted when a crosschain transfer burns tokens.
    /// @param from     Address of the account tokens are being burned from.
    /// @param amount   Amount of tokens burned.
    /// @param sender   Address of the account that initiated the crosschain transfer.
    event CrosschainBurn(address indexed from, uint256 amount, address indexed sender);
    /// @notice Mint tokens through a crosschain transfer.
    /// @param _to     Address to mint tokens to.
    /// @param _amount Amount of tokens to mint.
    function crosschainMint(address _to, uint256 _amount) external;
    /// @notice Burn tokens through a crosschain transfer.
    /// @param _from   Address to burn tokens from.
    /// @param _amount Amount of tokens to burn.
    function crosschainBurn(address _from, uint256 _amount) external;
}


// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
/// @title Predeploys
/// @notice Contains constant addresses for protocol contracts that are pre-deployed to the L2
system.
//          This excludes the preinstalls (non-protocol contracts).
library Predeploys {
    /// @notice Number of predeploy-namespace addresses reserved for protocol usage.
    uint256 internal constant PREDEPLOY_COUNT = 2048;
    /// @custom:legacy
    /// @notice Address of the LegacyMessagePasser predeploy. Deprecate. Use the updated
    ///         L2ToL1MessagePasser contract instead.
    address internal constant LEGACY_MESSAGE_PASSER =
0x4200000000000000000000000000000000000000;
    /// @custom:legacy
```

```
    /// @notice Address of the L1MessageSender predeploy. Deprecated. Use
L2CrossDomainMessenger
    ///         or access tx.origin (or msg.sender) in a L1 to L2 transaction instead.
    ///         Not embedded into new OP-Stack chains.
    address internal constant L1_MESSAGE_SENDER =
0x4200000000000000000000000000000000000001;
    /// @custom:legacy
    /// @notice Address of the DeployerWhitelist predeploy. No longer active.
    address internal constant DEPLOYER_WHITELIST =
0x4200000000000000000000000000000000000002;
    /// @notice Address of the canonical WETH contract.
    address internal constant WETH = 0x4200000000000000000000000000000000000006;
    /// @notice Address of the L2CrossDomainMessenger predeploy.
    address internal constant L2_CROSS_DOMAIN_MESSENGER =
0x4200000000000000000000000000000000000007;
    /// @notice Address of the GasPriceOracle predeploy. Includes fee information
    ///         and helpers for computing the L1 portion of the transaction fee.
    address internal constant GAS_PRICE_ORACLE =
0x420000000000000000000000000000000000000F;
    /// @notice Address of the L2StandardBridge predeploy.
    address internal constant L2_STANDARD_BRIDGE =
0x4200000000000000000000000000000000000010;
    //// @notice Address of the SequencerFeeWallet predeploy.
    address internal constant SEQUENCER_FEE_WALLET =
0x4200000000000000000000000000000000000011;
    /// @notice Address of the OptimismMintableERC20Factory predeploy.
    address internal constant OPTIMISM_MINTABLE_ERC20_FACTORY =
0x4200000000000000000000000000000000000012;
    /// @custom:legacy
    /// @notice Address of the L1BlockNumber predeploy. Deprecated. Use the L1Block
predeploy
    ///         instead, which exposes more information about the L1 state.
    address internal constant L1_BLOCK_NUMBER =
0x4200000000000000000000000000000000000013;
    /// @notice Address of the L2ERC721Bridge predeploy.
    address internal constant L2_ERC721_BRIDGE =
0x4200000000000000000000000000000000000014;
    /// @notice Address of the L1Block predeploy.
    address internal constant L1_BLOCK_ATTRIBUTES =
0x4200000000000000000000000000000000000015;
    /// @notice Address of the L2ToL1MessagePasser predeploy.
    address internal constant L2_TO_L1_MESSAGE_PASSER =
0x4200000000000000000000000000000000000016;
    /// @notice Address of the OptimismMintableERC721Factory predeploy.
```

```
    address internal constant OPTIMISM_MINTABLE_ERC721_FACTORY =
0x4200000000000000000000000000000000000017;
    /// @notice Address of the ProxyAdmin predeploy.
    address internal constant PROXY_ADMIN =
0x4200000000000000000000000000000000000018;
    /// @notice Address of the BaseFeeVault predeploy.
    address internal constant BASE_FEE_VAULT =
0x4200000000000000000000000000000000000019;
    /// @notice Address of the L1FeeVault predeploy.
    address internal constant L1_FEE_VAULT =
0x420000000000000000000000000000000000001A;
    /// @notice Address of the SchemaRegistry predeploy.
    address internal constant SCHEMA_REGISTRY =
0x4200000000000000000000000000000000000020;
    /// @notice Address of the EAS predeploy.
    address internal constant EAS = 0x4200000000000000000000000000000000000021;
    /// @notice Address of the GovernanceToken predeploy.
    address internal constant GOVERNANCE_TOKEN =
0x4200000000000000000000000000000000000042;
    /// @custom:legacy
    /// @notice Address of the LegacyERC20ETH predeploy. Deprecated. Balances are migrated
to the
    ///         state trie as of the Bedrock upgrade. Contract has been locked and write functions
    ///         can no longer be accessed.
    address internal constant LEGACY_ERC20_ETH =
0xDeadDeAddeAddEAddeadDEaDDEAdDeaDDeAD0000;
    /// @notice Address of the CrossL2Inbox predeploy.
    address internal constant CROSS_L2_INBOX =
0x4200000000000000000000000000000000000022;
    /// @notice Address of the L2ToL2CrossDomainMessenger predeploy.
    address internal constant L2_TO_L2_CROSS_DOMAIN_MESSENGER =
0x4200000000000000000000000000000000000023;
    /// @notice Address of the SuperchainWETH predeploy.
    address internal constant SUPERCHAIN_WETH =
0x4200000000000000000000000000000000000024;
    /// @notice Address of the ETHLiquidity predeploy.
    address internal constant ETH_LIQUIDITY =
0x4200000000000000000000000000000000000025;
    /// @notice Address of the OptimismSuperchainERC20Factory predeploy.
    address internal constant OPTIMISM_SUPERCHAIN_ERC20_FACTORY =
0x4200000000000000000000000000000000000026;
    /// @notice Address of the OptimismSuperchainERC20Beacon predeploy.
    address internal constant OPTIMISM_SUPERCHAIN_ERC20_BEACON =
0x4200000000000000000000000000000000000027;
```

```
    // TODO: Precalculate the address of the implementation contract
    /// @notice Arbitrary address of the OptimismSuperchainERC20 implementation contract.
    address internal constant OPTIMISM_SUPERCHAIN_ERC20 =
0xB9415c6cA93bdC545D4c5177512FCC22EFa38F28;
    /// @notice Address of the SuperchainTokenBridge predeploy.
    address internal constant SUPERCHAIN_TOKEN_BRIDGE =
0x4200000000000000000000000000000000000028;
    /// @notice Returns the name of the predeploy at the given address.
    function getName(address _addr) internal pure returns (string memory out_) {
        require(isPredeployNamespace(_addr), "Predeploys: address must be a predeploy");
        if (_addr == LEGACY_MESSAGE_PASSER) return "LegacyMessagePasser";
        if (_addr == L1_MESSAGE_SENDER) return "L1MessageSender";
        if (_addr == DEPLOYER_WHITELIST) return "DeployerWhitelist";
        if (_addr == WETH) return "WETH";
        if (_addr == L2_CROSS_DOMAIN_MESSENGER) return "L2CrossDomainMessenger";
        if (_addr == GAS_PRICE_ORACLE) return "GasPriceOracle";
        if (_addr == L2_STANDARD_BRIDGE) return "L2StandardBridge";
        if (_addr == SEQUENCER_FEE_WALLET) return "SequencerFeeVault";
        if (_addr == OPTIMISM_MINTABLE_ERC20_FACTORY) return
"OptimismMintableERC20Factory";
        if (_addr == L1_BLOCK_NUMBER) return "L1BlockNumber";
        if (_addr == L2_ERC721_BRIDGE) return "L2ERC721Bridge";
        if (_addr == L1_BLOCK_ATTRIBUTES) return "L1Block";
        if (_addr == L2_TO_L1_MESSAGE_PASSER) return "L2ToL1MessagePasser";
        if (_addr == OPTIMISM_MINTABLE_ERC721_FACTORY) return
"OptimismMintableERC721Factory";
        if (_addr == PROXY_ADMIN) return "ProxyAdmin";
        if (_addr == BASE_FEE_VAULT) return "BaseFeeVault";
        if (_addr == L1_FEE_VAULT) return "L1FeeVault";
        if (_addr == SCHEMA_REGISTRY) return "SchemaRegistry";
        if (_addr == EAS) return "EAS";
        if (_addr == GOVERNANCE_TOKEN) return "GovernanceToken";
        if (_addr == LEGACY_ERC20_ETH) return "LegacyERC20ETH";
        if (_addr == CROSS_L2_INBOX) return "CrossL2Inbox";
        if (_addr == L2_TO_L2_CROSS_DOMAIN_MESSENGER) return
"L2ToL2CrossDomainMessenger";
        if (_addr == SUPERCHAIN_WETH) return "SuperchainWETH";
        if (_addr == ETH_LIQUIDITY) return "ETHLiquidity";
        if (_addr == OPTIMISM_SUPERCHAIN_ERC20_FACTORY) return
"OptimismSuperchainERC20Factory";
        if (_addr == OPTIMISM_SUPERCHAIN_ERC20_BEACON) return
"OptimismSuperchainERC20Beacon";
        if (_addr == SUPERCHAIN_TOKEN_BRIDGE) return "SuperchainTokenBridge";
        revert("Predeploys: unnamed predeploy");
```

```
    }
    /// @notice Returns true if the predeploy is not proxied.
    function notProxied(address _addr) internal pure returns (bool) {
        return _addr == GOVERNANCE_TOKEN || _addr == WETH;
    }
    /// @notice Returns true if the address is a defined predeploy that is embedded into new
OP-Stack chains.
    function isSupportedPredeploy(address _addr, bool _useInterop) internal pure returns (bool) {
        return _addr == LEGACY_MESSAGE_PASSER || _addr == DEPLOYER_WHITELIST ||
_addr == WETH
            || _addr == L2_CROSS_DOMAIN_MESSENGER || _addr == GAS_PRICE_ORACLE ||
_addr == L2_STANDARD_BRIDGE
            || _addr == SEQUENCER_FEE_WALLET || _addr ==
OPTIMISM_MINTABLE_ERC20_FACTORY || _addr == L1_BLOCK_NUMBER
            || _addr == L2_ERC721_BRIDGE || _addr == L1_BLOCK_ATTRIBUTES || _addr ==
L2_TO_L1_MESSAGE_PASSER
            || _addr == OPTIMISM_MINTABLE_ERC721_FACTORY || _addr == PROXY_ADMIN ||
_addr == BASE_FEE_VAULT
            || _addr == L1_FEE_VAULT || _addr == SCHEMA_REGISTRY || _addr == EAS || _addr
== GOVERNANCE_TOKEN
            || (_useInterop && _addr == CROSS_L2_INBOX) || (_useInterop && _addr ==
L2_TO_L2_CROSS_DOMAIN_MESSENGER)
            || (_useInterop && _addr == SUPERCHAIN_WETH) || (_useInterop && _addr ==
ETH_LIQUIDITY)
            || (_useInterop && _addr == OPTIMISM_SUPERCHAIN_ERC20_FACTORY)
            || (_useInterop && _addr == OPTIMISM_SUPERCHAIN_ERC20_BEACON)
            || (_useInterop && _addr == SUPERCHAIN_TOKEN_BRIDGE);
    }
    function isPredeployNamespace(address _addr) internal pure returns (bool) {
        return uint160(_addr) >> 11 ==
uint160(0x4200000000000000000000000000000000000000) >> 11;
    }
    /// @notice Function to compute the expected address of the predeploy implementation
    ///         in the genesis state.
    function predeployToCodeNamespace(address _addr) internal pure returns (address) {
        require(
            isPredeployNamespace(_addr), "Predeploys: can only derive code-namespace address
for predeploy addresses"
        );
        return address(
            uint160(uint256(uint160(_addr)) & 0xffff |
uint256(uint160(0xc0D3C0d3C0d3C0D3c0d3C0d3c0D3C0d3c0d30000)))
        );
    }
```

```
}


// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
/// @notice Error for an unauthorized CALLER.
error Unauthorized();
/// @notice Error for when a method is called that only works when using a custom gas token.
error OnlyCustomGasToken();
/// @notice Error for when a method is called that only works when NOT using a custom gas
token.
error NotCustomGasToken();
/// @notice Error for when a transfer via call fails.
error TransferFailed();
/// @notice Thrown when attempting to perform an operation and the account is the zero
address.
error ZeroAddress();


// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts (last updated v5.1.0) (token/ERC20/extensions/IERC20Metadata.sol)
pragma solidity ^0.8.20;
import {IERC20} from "../IERC20.sol";
/**
 * @dev Interface for the optional metadata functions from the ERC-20 standard.
 */
interface IERC20Metadata is IERC20 {
    /**
     * @dev Returns the name of the token.
     */
    function name() external view returns (string memory);
    /**
     * @dev Returns the symbol of the token.
     */
    function symbol() external view returns (string memory);
    /**
     * @dev Returns the decimals places of the token.
     */
    function decimals() external view returns (uint8);
}


// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts (last updated v5.0.1) (utils/Context.sol)
```

```
pragma solidity ^0.8.20;
/**
 * @dev Provides information about the current execution context, including the
 * sender of the transaction and its data. While these are generally available
 * via msg.sender and msg.data, they should not be accessed in such a direct
 * manner, since when dealing with meta-transactions the account sending and
 * paying for execution may not be the actual sender (as far as an application
 * is concerned).
 *
 * This contract is only required for intermediate, library-like contracts.
 */
abstract contract Context {
    function _msgSender() internal view virtual returns (address) {
        return msg.sender;
    }
    function _msgData() internal view virtual returns (bytes calldata) {
        return msg.data;
    }
    function _contextSuffixLength() internal view virtual returns (uint256) {
        return 0;
    }
}


// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts (last updated v5.1.0) (interfaces/draft-IERC6093.sol)
pragma solidity ^0.8.20;
/**
 * @dev Standard ERC-20 Errors
 * Interface of the https://eips.ethereum.org/EIPS/eip-6093[ERC-6093] custom errors for ERC-20
tokens.
 */
interface IERC20Errors {
    /**
     * @dev Indicates an error related to the current `balance` of a `sender`. Used in transfers.
     * @param sender Address whose tokens are being transferred.
     * @param balance Current balance for the interacting account.
     * @param needed Minimum amount required to perform a transfer.
     */
    error ERC20InsufficientBalance(address sender, uint256 balance, uint256 needed);
    /**
     * @dev Indicates a failure with the token `sender`. Used in transfers.
     * @param sender Address whose tokens are being transferred.
     */
```

```
    error ERC20InvalidSender(address sender);
    /**
     * @dev Indicates a failure with the token `receiver`. Used in transfers.
     * @param receiver Address to which tokens are being transferred.
     */
    error ERC20InvalidReceiver(address receiver);
    /**
     * @dev Indicates a failure with the `spender`’s `allowance`. Used in transfers.
     * @param spender Address that may be allowed to operate on tokens without being their owner.
     * @param allowance Amount of tokens a `spender` is allowed to operate with.
     * @param needed Minimum amount required to perform a transfer.
     */
    error ERC20InsufficientAllowance(address spender, uint256 allowance, uint256 needed);
    /**
     * @dev Indicates a failure with the `approver` of a token to be approved. Used in approvals.
     * @param approver Address initiating an approval operation.
     */
    error ERC20InvalidApprover(address approver);
    /**
     * @dev Indicates a failure with the `spender` to be approved. Used in approvals.
     * @param spender Address that may be allowed to operate on tokens without being their owner.
     */
    error ERC20InvalidSpender(address spender);
}
/**
 * @dev Standard ERC-721 Errors
 * Interface of the https://eips.ethereum.org/EIPS/eip-6093[ERC-6093] custom errors for ERC-721 tokens.
 */
interface IERC721Errors {
    /**
     * @dev Indicates that an address can't be an owner. For example, `address(0)` is a forbidden owner in ERC-20.
     * Used in balance queries.
     * @param owner Address of the current owner of a token.
     */
    error ERC721InvalidOwner(address owner);
    /**
     * @dev Indicates a `tokenId` whose `owner` is the zero address.
     * @param tokenId Identifier number of a token.
     */
    error ERC721NonexistentToken(uint256 tokenId);
```

```
    /**
     * @dev Indicates an error related to the ownership over a particular token. Used in transfers.
     * @param sender Address whose tokens are being transferred.
     * @param tokenId Identifier number of a token.
     * @param owner Address of the current owner of a token.
     */
    error ERC721IncorrectOwner(address sender, uint256 tokenId, address owner);
    /**
     * @dev Indicates a failure with the token `sender`. Used in transfers.
     * @param sender Address whose tokens are being transferred.
     */
    error ERC721InvalidSender(address sender);
    /**
     * @dev Indicates a failure with the token `receiver`. Used in transfers.
     * @param receiver Address to which tokens are being transferred.
     */
    error ERC721InvalidReceiver(address receiver);
    /**
     * @dev Indicates a failure with the `operator`’s approval. Used in transfers.
     * @param operator Address that may be allowed to operate on tokens without being their
owner.
     * @param tokenId Identifier number of a token.
     */
    error ERC721InsufficientApproval(address operator, uint256 tokenId);
    /**
     * @dev Indicates a failure with the `approver` of a token to be approved. Used in approvals.
     * @param approver Address initiating an approval operation.
     */
    error ERC721InvalidApprover(address approver);
    /**
     * @dev Indicates a failure with the `operator` to be approved. Used in approvals.
     * @param operator Address that may be allowed to operate on tokens without being their
owner.
     */
    error ERC721InvalidOperator(address operator);
}
/**
 * @dev Standard ERC-1155 Errors
 * Interface of the https://eips.ethereum.org/EIPS/eip-6093[ERC-6093] custom errors for
ERC-1155 tokens.
 */
interface IERC1155Errors {
    /**
     * @dev Indicates an error related to the current `balance` of a `sender`. Used in transfers.
```

```
     * @param sender Address whose tokens are being transferred.
     * @param balance Current balance for the interacting account.
     * @param needed Minimum amount required to perform a transfer.
     * @param tokenId Identifier number of a token.
     */
    error ERC1155InsufficientBalance(address sender, uint256 balance, uint256 needed, uint256
tokenId);
    /**
     * @dev Indicates a failure with the token `sender`. Used in transfers.
     * @param sender Address whose tokens are being transferred.
     */
    error ERC1155InvalidSender(address sender);
    /**
     * @dev Indicates a failure with the token `receiver`. Used in transfers.
     * @param receiver Address to which tokens are being transferred.
     */
    error ERC1155InvalidReceiver(address receiver);
    /**
     * @dev Indicates a failure with the `operator`'s approval. Used in transfers.
     * @param operator Address that may be allowed to operate on tokens without being their
owner.
     * @param owner Address of the current owner of a token.
     */
    error ERC1155MissingApprovalForAll(address operator, address owner);
    /**
     * @dev Indicates a failure with the `approver` of a token to be approved. Used in approvals.
     * @param approver Address initiating an approval operation.
     */
    error ERC1155InvalidApprover(address approver);
    /**
     * @dev Indicates a failure with the `operator` to be approved. Used in approvals.
     * @param operator Address that may be allowed to operate on tokens without being their
owner.
     */
    error ERC1155InvalidOperator(address operator);
    /**
     * @dev Indicates an array length mismatch between ids and values in a
safeBatchTransferFrom operation.
     * Used in batch transfers.
     * @param idsLength Length of the array of token identifiers
     * @param valuesLength Length of the array of token amounts
     */
    error ERC1155InvalidArrayLength(uint256 idsLength, uint256 valuesLength);
}
```

```
// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts (last updated v5.1.0) (utils/introspection/IERC165.sol)
pragma solidity ^0.8.20;
/**
 * @dev Interface of the ERC-165 standard, as defined in the
 * https://eips.ethereum.org/EIPS/eip-165[ERC].
 *
 * Implementers can declare support of contract interfaces, which can then be
 * queried by others ({ERC165Checker}).
 *
 * For an implementation, see {ERC165}.
 */
interface IERC165 {
    /**
     * @dev Returns true if this contract implements the interface defined by
     * `interfaceId`. See the corresponding
     * https://eips.ethereum.org/EIPS/eip-165#how-interfaces-are-identified[ERC section]
     * to learn more about how these ids are created.
     *
     * This function call must use less than 30 000 gas.
     */
    function supportsInterface(bytes4 interfaceId) external view returns (bool);
}


// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts (last updated v5.1.0) (governance/utils/Votes.sol)
pragma solidity ^0.8.20;
import {IERC5805} from "../../interfaces/IERC5805.sol";
import {Context} from "../../utils/Context.sol";
import {Nonces} from "../../utils/Nonces.sol";
import {EIP712} from "../../utils/cryptography/EIP712.sol";
import {Checkpoints} from "../../utils/structs/Checkpoints.sol";
import {SafeCast} from "../../utils/math/SafeCast.sol";
import {ECDSA} from "../../utils/cryptography/ECDSA.sol";
import {Time} from "../../utils/types/Time.sol";
/**
 * @dev This is a base abstract contract that tracks voting units, which are a measure of voting
power that can be
 * transferred, and provides a system of vote delegation, where an account can delegate its
voting units to a sort of
```

```
 * "representative" that will pool delegated voting units from different accounts and can then use it to vote in
 * decisions. In fact, voting units _must_ be delegated in order to count as actual votes, and an account has to
 * delegate those votes to itself if it wishes to participate in decisions and does not have a trusted representative.
 *
 * This contract is often combined with a token contract such that voting units correspond to token units. For an
 * example, see {ERC721Votes}.
 *
 * The full history of delegate votes is tracked on-chain so that governance protocols can consider votes as distributed
 * at a particular block number to protect against flash loans and double voting. The opt-in delegate system makes the
 * cost of this history tracking optional.
 *
 * When using this module the derived contract must implement {_getVotingUnits} (for example, make it return
 * {ERC721-balanceOf}), and can use {_transferVotingUnits} to track a change in the distribution of those units (in the
 * previous example, it would be included in {ERC721-_update}).
 */
abstract contract Votes is Context, EIP712, Nonces, IERC5805 {
    using Checkpoints for Checkpoints.Trace208;
    bytes32 private constant DELEGATION_TYPEHASH =
        keccak256("Delegation(address delegatee,uint256 nonce,uint256 expiry)");
    mapping(address account => address) private _delegatee;
    mapping(address delegatee => Checkpoints.Trace208) private _delegateCheckpoints;
    Checkpoints.Trace208 private _totalCheckpoints;
    /**
     * @dev The clock was incorrectly modified.
     */
    error ERC6372InconsistentClock();
    /**
     * @dev Lookup to future votes is not available.
     */
    error ERC5805FutureLookup(uint256 timepoint, uint48 clock);
    /**
     * @dev Clock used for flagging checkpoints. Can be overridden to implement timestamp based
     * checkpoints (and voting), in which case {CLOCK_MODE} should be overridden as well to match.
     */
```

```
    function clock() public view virtual returns (uint48) {
        return Time.blockNumber();
    }
    /**
     * @dev Machine-readable description of the clock as specified in ERC-6372.
     */
    // solhint-disable-next-line func-name-mixedcase
    function CLOCK_MODE() public view virtual returns (string memory) {
        // Check that the clock was not modified
        if (clock() != Time.blockNumber()) {
            revert ERC6372InconsistentClock();
        }
        return "mode=blocknumber&from=default";
    }
    /**
     * @dev Returns the current amount of votes that `account` has.
     */
    function getVotes(address account) public view virtual returns (uint256) {
        return _delegateCheckpoints[account].latest();
    }
    /**
     * @dev Returns the amount of votes that `account` had at a specific moment in the past. If the `clock()` is
     * configured to use block numbers, this will return the value at the end of the corresponding block.
     *
     * Requirements:
     *
     * - `timepoint` must be in the past. If operating using block numbers, the block must be already mined.
     */
    function getPastVotes(address account, uint256 timepoint) public view virtual returns (uint256) {
        uint48 currentTimepoint = clock();
        if (timepoint >= currentTimepoint) {
            revert ERC5805FutureLookup(timepoint, currentTimepoint);
        }
        return _delegateCheckpoints[account].upperLookupRecent(SafeCast.toUint48(timepoint));
    }
    /**
     * @dev Returns the total supply of votes available at a specific moment in the past. If the `clock()` is
     * configured to use block numbers, this will return the value at the end of the corresponding block.
```

```
     *
     * NOTE: This value is the sum of all available votes, which is not necessarily the sum of all
delegated votes.
     * Votes that have not been delegated are still part of total supply, even though they would not
participate in a
     * vote.
     *
     * Requirements:
     *
     * - `timepoint` must be in the past. If operating using block numbers, the block must be
already mined.
     */
    function getPastTotalSupply(uint256 timepoint) public view virtual returns (uint256) {
        uint48 currentTimepoint = clock();
        if (timepoint >= currentTimepoint) {
            revert ERC5805FutureLookup(timepoint, currentTimepoint);
        }
        return _totalCheckpoints.upperLookupRecent(SafeCast.toUint48(timepoint));
    }
    /**
     * @dev Returns the current total supply of votes.
     */
    function _getTotalSupply() internal view virtual returns (uint256) {
        return _totalCheckpoints.latest();
    }
    /**
     * @dev Returns the delegate that `account` has chosen.
     */
    function delegates(address account) public view virtual returns (address) {
        return _delegatee[account];
    }
    /**
     * @dev Delegates votes from the sender to `delegatee`.
     */
    function delegate(address delegatee) public virtual {
        address account = _msgSender();
        _delegate(account, delegatee);
    }
    /**
     * @dev Delegates votes from signer to `delegatee`.
     */
    function delegateBySig(
        address delegatee,
        uint256 nonce,
```

```
        uint256 expiry,
        uint8 v,
        bytes32 r,
        bytes32 s
    ) public virtual {
        if (block.timestamp > expiry) {
            revert VotesExpiredSignature(expiry);
        }
        address signer = ECDSA.recover(
            _hashTypedDataV4(keccak256(abi.encode(DELEGATION_TYPEHASH, delegatee,
nonce, expiry))),
            v,
            r,
            s
        );
        _useCheckedNonce(signer, nonce);
        _delegate(signer, delegatee);
    }
    /**
     * @dev Delegate all of `account`'s voting units to `delegatee`.
     *
     * Emits events {IVotes-DelegateChanged} and {IVotes-DelegateVotesChanged}.
     */
    function _delegate(address account, address delegatee) internal virtual {
        address oldDelegate = delegates(account);
        _delegatee[account] = delegatee;
        emit DelegateChanged(account, oldDelegate, delegatee);
        _moveDelegateVotes(oldDelegate, delegatee, _getVotingUnits(account));
    }
    /**
     * @dev Transfers, mints, or burns voting units. To register a mint, `from` should be zero. To
register a burn, `to`
     * should be zero. Total supply of voting units will be adjusted with mints and burns.
     */
    function _transferVotingUnits(address from, address to, uint256 amount) internal virtual {
        if (from == address(0)) {
            _push(_totalCheckpoints, _add, SafeCast.toUint208(amount));
        }
        if (to == address(0)) {
            _push(_totalCheckpoints, _subtract, SafeCast.toUint208(amount));
        }
        _moveDelegateVotes(delegates(from), delegates(to), amount);
    }
    /**
```

```
 * @dev Moves delegated votes from one delegate to another.
 */
function _moveDelegateVotes(address from, address to, uint256 amount) internal virtual {
    if (from != to && amount > 0) {
        if (from != address(0)) {
            (uint256 oldValue, uint256 newValue) = _push(
                _delegateCheckpoints[from],
                _subtract,
                SafeCast.toUint208(amount)
            );
            emit DelegateVotesChanged(from, oldValue, newValue);
        }
        if (to != address(0)) {
            (uint256 oldValue, uint256 newValue) = _push(
                _delegateCheckpoints[to],
                _add,
                SafeCast.toUint208(amount)
            );
            emit DelegateVotesChanged(to, oldValue, newValue);
        }
    }
}
/**
 * @dev Get number of checkpoints for `account`.
 */
function _numCheckpoints(address account) internal view virtual returns (uint32) {
    return SafeCast.toUint32(_delegateCheckpoints[account].length());
}
/**
 * @dev Get the `pos`-th checkpoint for `account`.
 */
function _checkpoints(
    address account,
    uint32 pos
) internal view virtual returns (Checkpoints.Checkpoint208 memory) {
    return _delegateCheckpoints[account].at(pos);
}
function _push(
    Checkpoints.Trace208 storage store,
    function(uint208, uint208) view returns (uint208) op,
    uint208 delta
) private returns (uint208 oldValue, uint208 newValue) {
    return store.push(clock(), op(store.latest(), delta));
}
```

```
    function _add(uint208 a, uint208 b) private pure returns (uint208) {
        return a + b;
    }
    function _subtract(uint208 a, uint208 b) private pure returns (uint208) {
        return a - b;
    }
    /**
     * @dev Must return the voting units held by an account.
     */
    function _getVotingUnits(address) internal view virtual returns (uint256);
}


// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts (last updated v5.1.0) (utils/structs/Checkpoints.sol)
// This file was procedurally generated from scripts/generate/templates/Checkpoints.js.
pragma solidity ^0.8.20;
import {Math} from "../math/Math.sol";
/**
 * @dev This library defines the `Trace*` struct, for checkpointing values as they change at
different points in
 * time, and later looking up past values by block number. See {Votes} as an example.
 *
 * To create a history of checkpoints define a variable type `Checkpoints.Trace*` in your
contract, and store a new
 * checkpoint for the current transaction block using the {push} function.
 */
library Checkpoints {
    /**
     * @dev A value was attempted to be inserted on a past checkpoint.
     */
    error CheckpointUnorderedInsertion();
    struct Trace224 {
        Checkpoint224[] _checkpoints;
    }
    struct Checkpoint224 {
        uint32 _key;
        uint224 _value;
    }
    /**
     * @dev Pushes a (`key`, `value`) pair into a Trace224 so that it is stored as the checkpoint.
     *
     * Returns previous value and new value.
     *
```

```
     * IMPORTANT: Never accept `key` as a user input, since an arbitrary `type(uint32).max` key
set will disable the
     * library.
     */
    function push(
        Trace224 storage self,
        uint32 key,
        uint224 value
    ) internal returns (uint224 oldValue, uint224 newValue) {
        return _insert(self._checkpoints, key, value);
    }
    /**
     * @dev Returns the value in the first (oldest) checkpoint with key greater or equal than the
search key, or zero if
     * there is none.
     */
    function lowerLookup(Trace224 storage self, uint32 key) internal view returns (uint224) {
        uint256 len = self._checkpoints.length;
        uint256 pos = _lowerBinaryLookup(self._checkpoints, key, 0, len);
        return pos == len ? 0 : _unsafeAccess(self._checkpoints, pos)._value;
    }
    /**
     * @dev Returns the value in the last (most recent) checkpoint with key lower or equal than
the search key, or zero
     * if there is none.
     */
    function upperLookup(Trace224 storage self, uint32 key) internal view returns (uint224) {
        uint256 len = self._checkpoints.length;
        uint256 pos = _upperBinaryLookup(self._checkpoints, key, 0, len);
        return pos == 0 ? 0 : _unsafeAccess(self._checkpoints, pos - 1)._value;
    }
    /**
     * @dev Returns the value in the last (most recent) checkpoint with key lower or equal than
the search key, or zero
     * if there is none.
     *
     * NOTE: This is a variant of {upperLookup} that is optimised to find "recent" checkpoint
(checkpoints with high
     * keys).
     */
    function upperLookupRecent(Trace224 storage self, uint32 key) internal view returns
(uint224) {
        uint256 len = self._checkpoints.length;
        uint256 low = 0;
```

```
        uint256 high = len;
        if (len > 5) {
            uint256 mid = len - Math.sqrt(len);
            if (key < _unsafeAccess(self._checkpoints, mid)._key) {
                high = mid;
            } else {
                low = mid + 1;
            }
        }
        uint256 pos = _upperBinaryLookup(self._checkpoints, key, low, high);
        return pos == 0 ? 0 : _unsafeAccess(self._checkpoints, pos - 1)._value;
    }
    /**
     * @dev Returns the value in the most recent checkpoint, or zero if there are no checkpoints.
     */
    function latest(Trace224 storage self) internal view returns (uint224) {
        uint256 pos = self._checkpoints.length;
        return pos == 0 ? 0 : _unsafeAccess(self._checkpoints, pos - 1)._value;
    }
    /**
     * @dev Returns whether there is a checkpoint in the structure (i.e. it is not empty), and if so
the key and value
     * in the most recent checkpoint.
     */
    function latestCheckpoint(Trace224 storage self) internal view returns (bool exists, uint32
_key, uint224 _value) {
        uint256 pos = self._checkpoints.length;
        if (pos == 0) {
            return (false, 0, 0);
        } else {
            Checkpoint224 storage ckpt = _unsafeAccess(self._checkpoints, pos - 1);
            return (true, ckpt._key, ckpt._value);
        }
    }
    /**
     * @dev Returns the number of checkpoint.
     */
    function length(Trace224 storage self) internal view returns (uint256) {
        return self._checkpoints.length;
    }
    /**
     * @dev Returns checkpoint at given position.
     */
    function at(Trace224 storage self, uint32 pos) internal view returns (Checkpoint224 memory) {
```

```
        return self._checkpoints[pos];
    }
    /**
     * @dev Pushes a (`key`, `value`) pair into an ordered list of checkpoints, either by inserting a new checkpoint,
     * or by updating the last one.
     */
    function _insert(
        Checkpoint224[] storage self,
        uint32 key,
        uint224 value
    ) private returns (uint224 oldValue, uint224 newValue) {
        uint256 pos = self.length;
        if (pos > 0) {
            Checkpoint224 storage last = _unsafeAccess(self, pos - 1);
            uint32 lastKey = last._key;
            uint224 lastValue = last._value;
            // Checkpoint keys must be non-decreasing.
            if (lastKey > key) {
                revert CheckpointUnorderedInsertion();
            }
            // Update or push new checkpoint
            if (lastKey == key) {
                last._value = value;
            } else {
                self.push(Checkpoint224({_key: key, _value: value}));
            }
            return (lastValue, value);
        } else {
            self.push(Checkpoint224({_key: key, _value: value}));
            return (0, value);
        }
    }
    /**
     * @dev Return the index of the first (oldest) checkpoint with key strictly bigger than the search key, or `high`
     * if there is none. `low` and `high` define a section where to do the search, with inclusive `low` and exclusive
     * `high`.
     *
     * WARNING: `high` should not be greater than the array's length.
     */
    function _upperBinaryLookup(
        Checkpoint224[] storage self,
```

```
        uint32 key,
        uint256 low,
        uint256 high
    ) private view returns (uint256) {
        while (low < high) {
            uint256 mid = Math.average(low, high);
            if (_unsafeAccess(self, mid)._key > key) {
                high = mid;
            } else {
                low = mid + 1;
            }
        }
        return high;
    }
    /**
     * @dev Return the index of the first (oldest) checkpoint with key greater or equal than the search key, or `high`
     * if there is none. `low` and `high` define a section where to do the search, with inclusive `low` and exclusive
     * `high`.
     *
     * WARNING: `high` should not be greater than the array's length.
     */
    function _lowerBinaryLookup(
        Checkpoint224[] storage self,
        uint32 key,
        uint256 low,
        uint256 high
    ) private view returns (uint256) {
        while (low < high) {
            uint256 mid = Math.average(low, high);
            if (_unsafeAccess(self, mid)._key < key) {
                low = mid + 1;
            } else {
                high = mid;
            }
        }
        return high;
    }
    /**
     * @dev Access an element of the array without performing bounds check. The position is assumed to be within bounds.
     */
    function _unsafeAccess(
```

```
        Checkpoint224[] storage self,
        uint256 pos
    ) private pure returns (Checkpoint224 storage result) {
        assembly {
            mstore(0, self.slot)
            result.slot := add(keccak256(0, 0x20), pos)
        }
    }
    struct Trace208 {
        Checkpoint208[] _checkpoints;
    }
    struct Checkpoint208 {
        uint48 _key;
        uint208 _value;
    }
    /**
     * @dev Pushes a (`key`, `value`) pair into a Trace208 so that it is stored as the checkpoint.
     *
     * Returns previous value and new value.
     *
     * IMPORTANT: Never accept `key` as a user input, since an arbitrary `type(uint48).max` key
set will disable the
     * library.
     */
    function push(
        Trace208 storage self,
        uint48 key,
        uint208 value
    ) internal returns (uint208 oldValue, uint208 newValue) {
        return _insert(self._checkpoints, key, value);
    }
    /**
     * @dev Returns the value in the first (oldest) checkpoint with key greater or equal than the
search key, or zero if
     * there is none.
     */
    function lowerLookup(Trace208 storage self, uint48 key) internal view returns (uint208) {
        uint256 len = self._checkpoints.length;
        uint256 pos = _lowerBinaryLookup(self._checkpoints, key, 0, len);
        return pos == len ? 0 : _unsafeAccess(self._checkpoints, pos)._value;
    }
    /**
     * @dev Returns the value in the last (most recent) checkpoint with key lower or equal than
the search key, or zero
```

```
     * if there is none.
     */
    function upperLookup(Trace208 storage self, uint48 key) internal view returns (uint208) {
        uint256 len = self._checkpoints.length;
        uint256 pos = _upperBinaryLookup(self._checkpoints, key, 0, len);
        return pos == 0 ? 0 : _unsafeAccess(self._checkpoints, pos - 1)._value;
    }
    /**
     * @dev Returns the value in the last (most recent) checkpoint with key lower or equal than the search key, or zero
     * if there is none.
     *
     * NOTE: This is a variant of {upperLookup} that is optimised to find "recent" checkpoint (checkpoints with high
     * keys).
     */
    function upperLookupRecent(Trace208 storage self, uint48 key) internal view returns (uint208) {
        uint256 len = self._checkpoints.length;
        uint256 low = 0;
        uint256 high = len;
        if (len > 5) {
            uint256 mid = len - Math.sqrt(len);
            if (key < _unsafeAccess(self._checkpoints, mid)._key) {
                high = mid;
            } else {
                low = mid + 1;
            }
        }
        uint256 pos = _upperBinaryLookup(self._checkpoints, key, low, high);
        return pos == 0 ? 0 : _unsafeAccess(self._checkpoints, pos - 1)._value;
    }
    /**
     * @dev Returns the value in the most recent checkpoint, or zero if there are no checkpoints.
     */
    function latest(Trace208 storage self) internal view returns (uint208) {
        uint256 pos = self._checkpoints.length;
        return pos == 0 ? 0 : _unsafeAccess(self._checkpoints, pos - 1)._value;
    }
    /**
     * @dev Returns whether there is a checkpoint in the structure (i.e. it is not empty), and if so the key and value
     * in the most recent checkpoint.
     */
```

```
    function latestCheckpoint(Trace208 storage self) internal view returns (bool exists, uint48
_key, uint208 _value) {
        uint256 pos = self._checkpoints.length;
        if (pos == 0) {
            return (false, 0, 0);
        } else {
            Checkpoint208 storage ckpt = _unsafeAccess(self._checkpoints, pos - 1);
            return (true, ckpt._key, ckpt._value);
        }
    }
    /**
     * @dev Returns the number of checkpoint.
     */
    function length(Trace208 storage self) internal view returns (uint256) {
        return self._checkpoints.length;
    }
    /**
     * @dev Returns checkpoint at given position.
     */
    function at(Trace208 storage self, uint32 pos) internal view returns (Checkpoint208 memory) {
        return self._checkpoints[pos];
    }
    /**
     * @dev Pushes a (`key`, `value`) pair into an ordered list of checkpoints, either by inserting a
new checkpoint,
     * or by updating the last one.
     */
    function _insert(
        Checkpoint208[] storage self,
        uint48 key,
        uint208 value
    ) private returns (uint208 oldValue, uint208 newValue) {
        uint256 pos = self.length;
        if (pos > 0) {
            Checkpoint208 storage last = _unsafeAccess(self, pos - 1);
            uint48 lastKey = last._key;
            uint208 lastValue = last._value;
            // Checkpoint keys must be non-decreasing.
            if (lastKey > key) {
                revert CheckpointUnorderedInsertion();
            }
            // Update or push new checkpoint
            if (lastKey == key) {
                last._value = value;
```

```
            } else {
                self.push(Checkpoint208({_key: key, _value: value}));
            }
            return (lastValue, value);
        } else {
            self.push(Checkpoint208({_key: key, _value: value}));
            return (0, value);
        }
    }
    /**
     * @dev Return the index of the first (oldest) checkpoint with key strictly bigger than the search key, or `high`
     * if there is none. `low` and `high` define a section where to do the search, with inclusive `low` and exclusive
     * `high`.
     *
     * WARNING: `high` should not be greater than the array's length.
     */
    function _upperBinaryLookup(
        Checkpoint208[] storage self,
        uint48 key,
        uint256 low,
        uint256 high
    ) private view returns (uint256) {
        while (low < high) {
            uint256 mid = Math.average(low, high);
            if (_unsafeAccess(self, mid)._key > key) {
                high = mid;
            } else {
                low = mid + 1;
            }
        }
        return high;
    }
    /**
     * @dev Return the index of the first (oldest) checkpoint with key greater or equal than the search key, or `high`
     * if there is none. `low` and `high` define a section where to do the search, with inclusive `low` and exclusive
     * `high`.
     *
     * WARNING: `high` should not be greater than the array's length.
     */
    function _lowerBinaryLookup(
```

```
        Checkpoint208[] storage self,
        uint48 key,
        uint256 low,
        uint256 high
    ) private view returns (uint256) {
        while (low < high) {
            uint256 mid = Math.average(low, high);
            if (_unsafeAccess(self, mid)._key < key) {
                low = mid + 1;
            } else {
                high = mid;
            }
        }
        return high;
    }
    /**
     * @dev Access an element of the array without performing bounds check. The position is assumed to be within bounds.
     */
    function _unsafeAccess(
        Checkpoint208[] storage self,
        uint256 pos
    ) private pure returns (Checkpoint208 storage result) {
        assembly {
            mstore(0, self.slot)
            result.slot := add(keccak256(0, 0x20), pos)
        }
    }
    struct Trace160 {
        Checkpoint160[] _checkpoints;
    }
    struct Checkpoint160 {
        uint96 _key;
        uint160 _value;
    }
    /**
     * @dev Pushes a (`key`, `value`) pair into a Trace160 so that it is stored as the checkpoint.
     *
     * Returns previous value and new value.
     *
     * IMPORTANT: Never accept `key` as a user input, since an arbitrary `type(uint96).max` key set will disable the
     * library.
     */
```

```
    function push(
        Trace160 storage self,
        uint96 key,
        uint160 value
    ) internal returns (uint160 oldValue, uint160 newValue) {
        return _insert(self._checkpoints, key, value);
    }
    /**
     * @dev Returns the value in the first (oldest) checkpoint with key greater or equal than the
search key, or zero if
     * there is none.
     */
    function lowerLookup(Trace160 storage self, uint96 key) internal view returns (uint160) {
        uint256 len = self._checkpoints.length;
        uint256 pos = _lowerBinaryLookup(self._checkpoints, key, 0, len);
        return pos == len ? 0 : _unsafeAccess(self._checkpoints, pos)._value;
    }
    /**
     * @dev Returns the value in the last (most recent) checkpoint with key lower or equal than
the search key, or zero
     * if there is none.
     */
    function upperLookup(Trace160 storage self, uint96 key) internal view returns (uint160) {
        uint256 len = self._checkpoints.length;
        uint256 pos = _upperBinaryLookup(self._checkpoints, key, 0, len);
        return pos == 0 ? 0 : _unsafeAccess(self._checkpoints, pos - 1)._value;
    }
    /**
     * @dev Returns the value in the last (most recent) checkpoint with key lower or equal than
the search key, or zero
     * if there is none.
     *
     * NOTE: This is a variant of {upperLookup} that is optimised to find "recent" checkpoint
(checkpoints with high
     * keys).
     */
    function upperLookupRecent(Trace160 storage self, uint96 key) internal view returns
(uint160) {
        uint256 len = self._checkpoints.length;
        uint256 low = 0;
        uint256 high = len;
        if (len > 5) {
            uint256 mid = len - Math.sqrt(len);
            if (key < _unsafeAccess(self._checkpoints, mid)._key) {
```

```
            high = mid;
        } else {
            low = mid + 1;
        }
    }
    uint256 pos = _upperBinaryLookup(self._checkpoints, key, low, high);
    return pos == 0 ? 0 : _unsafeAccess(self._checkpoints, pos - 1)._value;
}
/**
 * @dev Returns the value in the most recent checkpoint, or zero if there are no checkpoints.
 */
function latest(Trace160 storage self) internal view returns (uint160) {
    uint256 pos = self._checkpoints.length;
    return pos == 0 ? 0 : _unsafeAccess(self._checkpoints, pos - 1)._value;
}
/**
 * @dev Returns whether there is a checkpoint in the structure (i.e. it is not empty), and if so the key and value
 * in the most recent checkpoint.
 */
function latestCheckpoint(Trace160 storage self) internal view returns (bool exists, uint96 _key, uint160 _value) {
    uint256 pos = self._checkpoints.length;
    if (pos == 0) {
        return (false, 0, 0);
    } else {
        Checkpoint160 storage ckpt = _unsafeAccess(self._checkpoints, pos - 1);
        return (true, ckpt._key, ckpt._value);
    }
}
/**
 * @dev Returns the number of checkpoint.
 */
function length(Trace160 storage self) internal view returns (uint256) {
    return self._checkpoints.length;
}
/**
 * @dev Returns checkpoint at given position.
 */
function at(Trace160 storage self, uint32 pos) internal view returns (Checkpoint160 memory) {
    return self._checkpoints[pos];
}
/**
```

```
    * @dev Pushes a (`key`, `value`) pair into an ordered list of checkpoints, either by inserting a new checkpoint,
    * or by updating the last one.
    */
    function _insert(
        Checkpoint160[] storage self,
        uint96 key,
        uint160 value
    ) private returns (uint160 oldValue, uint160 newValue) {
        uint256 pos = self.length;
        if (pos > 0) {
            Checkpoint160 storage last = _unsafeAccess(self, pos - 1);
            uint96 lastKey = last._key;
            uint160 lastValue = last._value;
            // Checkpoint keys must be non-decreasing.
            if (lastKey > key) {
                revert CheckpointUnorderedInsertion();
            }
            // Update or push new checkpoint
            if (lastKey == key) {
                last._value = value;
            } else {
                self.push(Checkpoint160({_key: key, _value: value}));
            }
            return (lastValue, value);
        } else {
            self.push(Checkpoint160({_key: key, _value: value}));
            return (0, value);
        }
    }
    /**
    * @dev Return the index of the first (oldest) checkpoint with key strictly bigger than the search key, or `high`
    * if there is none. `low` and `high` define a section where to do the search, with inclusive `low` and exclusive
    * `high`.
    *
    * WARNING: `high` should not be greater than the array's length.
    */
    function _upperBinaryLookup(
        Checkpoint160[] storage self,
        uint96 key,
        uint256 low,
        uint256 high
```

```
    ) private view returns (uint256) {
        while (low < high) {
            uint256 mid = Math.average(low, high);
            if (_unsafeAccess(self, mid)._key > key) {
                high = mid;
            } else {
                low = mid + 1;
            }
        }
        return high;
    }
    /**
     * @dev Return the index of the first (oldest) checkpoint with key greater or equal than the search key, or `high`
     * if there is none. `low` and `high` define a section where to do the search, with inclusive `low` and exclusive
     * `high`.
     *
     * WARNING: `high` should not be greater than the array's length.
     */
    function _lowerBinaryLookup(
        Checkpoint160[] storage self,
        uint96 key,
        uint256 low,
        uint256 high
    ) private view returns (uint256) {
        while (low < high) {
            uint256 mid = Math.average(low, high);
            if (_unsafeAccess(self, mid)._key < key) {
                low = mid + 1;
            } else {
                high = mid;
            }
        }
        return high;
    }
    /**
     * @dev Access an element of the array without performing bounds check. The position is assumed to be within bounds.
     */
    function _unsafeAccess(
        Checkpoint160[] storage self,
        uint256 pos
    ) private pure returns (Checkpoint160 storage result) {
```

```
        assembly {
            mstore(0, self.slot)
            result.slot := add(keccak256(0, 0x20), pos)
        }
    }
}


// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts (last updated v5.1.0) (token/ERC20/extensions/IERC20Permit.sol)
pragma solidity ^0.8.20;
/**
 * @dev Interface of the ERC-20 Permit extension allowing approvals to be made via signatures, as defined in
 * https://eips.ethereum.org/EIPS/eip-2612[ERC-2612].
 *
 * Adds the {permit} method, which can be used to change an account's ERC-20 allowance (see {IERC20-allowance}) by
 * presenting a message signed by the account. By not relying on {IERC20-approve}, the token holder account doesn't
 * need to send a transaction, and thus is not required to hold Ether at all.
 *
 * ==== Security Considerations
 *
 * There are two important considerations concerning the use of `permit`. The first is that a valid permit signature
 * expresses an allowance, and it should not be assumed to convey additional meaning. In particular, it should not be
 * considered as an intention to spend the allowance in any specific way. The second is that because permits have
 * built-in replay protection and can be submitted by anyone, they can be frontrun. A protocol that uses permits should
 * take this into consideration and allow a `permit` call to fail. Combining these two aspects, a pattern that may be
 * generally recommended is:
 *
 * ```solidity
 * function doThingWithPermit(..., uint256 value, uint256 deadline, uint8 v, bytes32 r, bytes32 s) public {
 *     try token.permit(msg.sender, address(this), value, deadline, v, r, s) {} catch {}
 *     doThing(..., value);
 * }
 *
 * function doThing(..., uint256 value) public {
```

```
 *     token.safeTransferFrom(msg.sender, address(this), value);
 *     ...
 * }
 * ```
 *
 * Observe that: 1) `msg.sender` is used as the owner, leaving no ambiguity as to the signer intent, and 2) the use of
 * `try/catch` allows the permit to fail and makes the code tolerant to frontrunning. (See also
 * {SafeERC20-safeTransferFrom}).
 *
 * Additionally, note that smart contract wallets (such as Argent or Safe) are not able to produce permit signatures, so
 * contracts should have entry points that don't rely on permit.
 */
interface IERC20Permit {
    /**
     * @dev Sets `value` as the allowance of `spender` over ``owner``'s tokens,
     * given ``owner``'s signed approval.
     *
     * IMPORTANT: The same issues {IERC20-approve} has related to transaction
     * ordering also apply here.
     *
     * Emits an {Approval} event.
     *
     * Requirements:
     *
     * - `spender` cannot be the zero address.
     * - `deadline` must be a timestamp in the future.
     * - `v`, `r` and `s` must be a valid `secp256k1` signature from `owner`
     * over the EIP712-formatted function arguments.
     * - the signature must use ``owner``'s current nonce (see {nonces}).
     *
     * For more information on the signature format, see the
     * https://eips.ethereum.org/EIPS/eip-2612#specification[relevant EIP
     * section].
     *
     * CAUTION: See Security Considerations above.
     */
    function permit(
        address owner,
        address spender,
        uint256 value,
        uint256 deadline,
        uint8 v,
```

```
        bytes32 r,
        bytes32 s
    ) external;
    /**
     * @dev Returns the current nonce for `owner`. This value must be
     * included whenever a signature is generated for {permit}.
     *
     * Every successful call to {permit} increases ``owner``'s nonce by one. This
     * prevents a signature from being used multiple times.
     */
    function nonces(address owner) external view returns (uint256);
    /**
     * @dev Returns the domain separator used in the encoding of the signature for {permit}, as
defined by {EIP712}.
     */
    // solhint-disable-next-line func-name-mixedcase
    function DOMAIN_SEPARATOR() external view returns (bytes32);
}


// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts (last updated v5.1.0) (utils/cryptography/ECDSA.sol)
pragma solidity ^0.8.20;
/**
 * @dev Elliptic Curve Digital Signature Algorithm (ECDSA) operations.
 *
 * These functions can be used to verify that a message was signed by the holder
 * of the private keys of a given address.
 */
library ECDSA {
    enum RecoverError {
        NoError,
        InvalidSignature,
        InvalidSignatureLength,
        InvalidSignatureS
    }
    /**
     * @dev The signature derives the `address(0)`.
     */
    error ECDSAInvalidSignature();
    /**
     * @dev The signature has an invalid length.
     */
    error ECDSAInvalidSignatureLength(uint256 length);
```

```
    /**
     * @dev The signature has an S value that is in the upper half order.
     */
    error ECDSAInvalidSignatureS(bytes32 s);
    /**
     * @dev Returns the address that signed a hashed message (`hash`) with `signature` or an
error. This will not
     * return address(0) without also returning an error description. Errors are documented using
an enum (error type)
     * and a bytes32 providing additional information about the error.
     *
     * If no error is returned, then the address can be used for verification purposes.
     *
     * The `ecrecover` EVM precompile allows for malleable (non-unique) signatures:
     * this function rejects them by requiring the `s` value to be in the lower
     * half order, and the `v` value to be either 27 or 28.
     *
     * IMPORTANT: `hash` _must_ be the result of a hash operation for the
     * verification to be secure: it is possible to craft signatures that
     * recover to arbitrary addresses for non-hashed data. A safe way to ensure
     * this is by receiving a hash of the original message (which may otherwise
     * be too long), and then calling {MessageHashUtils-toEthSignedMessageHash} on it.
     *
     * Documentation for signature generation:
     * - with https://web3js.readthedocs.io/en/v1.3.4/web3-eth-accounts.html#sign[Web3.js]
     * - with https://docs.ethers.io/v5/api/signer/#Signer-signMessage[ethers]
     */
    function tryRecover(
        bytes32 hash,
        bytes memory signature
    ) internal pure returns (address recovered, RecoverError err, bytes32 errArg) {
        if (signature.length == 65) {
            bytes32 r;
            bytes32 s;
            uint8 v;
            // ecrecover takes the signature parameters, and the only way to get them
            // currently is to use assembly.
            assembly ("memory-safe") {
                r := mload(add(signature, 0x20))
                s := mload(add(signature, 0x40))
                v := byte(0, mload(add(signature, 0x60)))
            }
            return tryRecover(hash, v, r, s);
        } else {
```

```
            return (address(0), RecoverError.InvalidSignatureLength, bytes32(signature.length));
        }
    }
    /**
     * @dev Returns the address that signed a hashed message (`hash`) with
     * `signature`. This address can then be used for verification purposes.
     *
     * The `ecrecover` EVM precompile allows for malleable (non-unique) signatures:
     * this function rejects them by requiring the `s` value to be in the lower
     * half order, and the `v` value to be either 27 or 28.
     *
     * IMPORTANT: `hash` _must_ be the result of a hash operation for the
     * verification to be secure: it is possible to craft signatures that
     * recover to arbitrary addresses for non-hashed data. A safe way to ensure
     * this is by receiving a hash of the original message (which may otherwise
     * be too long), and then calling {MessageHashUtils-toEthSignedMessageHash} on it.
     */
    function recover(bytes32 hash, bytes memory signature) internal pure returns (address) {
        (address recovered, RecoverError error, bytes32 errorArg) = tryRecover(hash, signature);
        _throwError(error, errorArg);
        return recovered;
    }
    /**
     * @dev Overload of {ECDSA-tryRecover} that receives the `r` and `vs` short-signature fields
separately.
     *
     * See https://eips.ethereum.org/EIPS/eip-2098[ERC-2098 short signatures]
     */
    function tryRecover(
        bytes32 hash,
        bytes32 r,
        bytes32 vs
    ) internal pure returns (address recovered, RecoverError err, bytes32 errArg) {
        unchecked {
            bytes32 s = vs & bytes32(0x7fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff);
            // We do not check for an overflow here since the shift operation results in 0 or 1.
            uint8 v = uint8((uint256(vs) >> 255) + 27);
            return tryRecover(hash, v, r, s);
        }
    }
    /**
     * @dev Overload of {ECDSA-recover} that receives the `r and `vs` short-signature fields
separately.
     */
```

```
    function recover(bytes32 hash, bytes32 r, bytes32 vs) internal pure returns (address) {
        (address recovered, RecoverError error, bytes32 errorArg) = tryRecover(hash, r, vs);
        _throwError(error, errorArg);
        return recovered;
    }
    /**
     * @dev Overload of {ECDSA-tryRecover} that receives the `v`,
     * `r` and `s` signature fields separately.
     */
    function tryRecover(
        bytes32 hash,
        uint8 v,
        bytes32 r,
        bytes32 s
    ) internal pure returns (address recovered, RecoverError err, bytes32 errArg) {
        // EIP-2 still allows signature malleability for ecrecover(). Remove this possibility and make
the signature
        // unique. Appendix F in the Ethereum Yellow paper
(https://ethereum.github.io/yellowpaper/paper.pdf), defines
        // the valid range for s in (301): 0 < s < secp256k1n ÷ 2 + 1, and for v in (302): v ∈ {27,
28}. Most
        // signatures from current libraries generate a unique signature with an s-value in the lower
half order.
        //
        // If your library generates malleable signatures, such as s-values in the upper range,
calculate a new s-value
        // with
0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141 - s1
and flip v from 27 to 28 or
        // vice versa. If your library also generates signatures with 0/1 for v instead 27/28, add 27
to v to accept
        // these malleable signatures as well.
        if (uint256(s) >
0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF5D576E7357A4501DDFE92F46681B20A0) {
            return (address(0), RecoverError.InvalidSignatureS, s);
        }
        // If the signature is valid (and not malleable), return the signer address
        address signer = ecrecover(hash, v, r, s);
        if (signer == address(0)) {
            return (address(0), RecoverError.InvalidSignature, bytes32(0));
        }
        return (signer, RecoverError.NoError, bytes32(0));
    }
    /**
```

```
     * @dev Overload of {ECDSA-recover} that receives the `v`,
     * `r` and `s` signature fields separately.
     */
    function recover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) internal pure returns (address) {
        (address recovered, RecoverError error, bytes32 errorArg) = tryRecover(hash, v, r, s);
        _throwError(error, errorArg);
        return recovered;
    }
    /**
     * @dev Optionally reverts with the corresponding custom error according to the `error`
argument provided.
     */
    function _throwError(RecoverError error, bytes32 errorArg) private pure {
        if (error == RecoverError.NoError) {
            return; // no error: do nothing
        } else if (error == RecoverError.InvalidSignature) {
            revert ECDSAInvalidSignature();
        } else if (error == RecoverError.InvalidSignatureLength) {
            revert ECDSAInvalidSignatureLength(uint256(errorArg));
        } else if (error == RecoverError.InvalidSignatureS) {
            revert ECDSAInvalidSignatureS(errorArg);
        }
    }
}


// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts (last updated v5.1.0) (utils/cryptography/EIP712.sol)
pragma solidity ^0.8.20;
import {MessageHashUtils} from "./MessageHashUtils.sol";
import {ShortStrings, ShortString} from "../ShortStrings.sol";
import {IERC5267} from "../../interfaces/IERC5267.sol";
/**
 * @dev https://eips.ethereum.org/EIPS/eip-712[EIP-712] is a standard for hashing and signing
of typed structured data.
 *
 * The encoding scheme specified in the EIP requires a domain separator and a hash of the
typed structured data, whose
 * encoding is very generic and therefore its implementation in Solidity is not feasible, thus this
contract
 * does not implement the encoding itself. Protocols need to implement the type-specific
encoding they need in order to
 * produce the hash of their typed data using a combination of `abi.encode` and `keccak256`.
 *
```

```
 * This contract implements the EIP-712 domain separator ({_domainSeparatorV4}) that is used as part of the encoding
 * scheme, and the final step of the encoding to obtain the message digest that is then signed via ECDSA
 * ({_hashTypedDataV4}).
 *
 * The implementation of the domain separator was designed to be as efficient as possible while still properly updating
 * the chain id to protect against replay attacks on an eventual fork of the chain.
 *
 * NOTE: This contract implements the version of the encoding known as "v4", as implemented by the JSON RPC method
 * https://docs.metamask.io/guide/signing-data.html[`eth_signTypedDataV4` in MetaMask].
 *
 * NOTE: In the upgradeable version of this contract, the cached values will correspond to the address, and the domain
 * separator of the implementation contract. This will cause the {_domainSeparatorV4} function to always rebuild the
 * separator from the immutable values, which is cheaper than accessing a cached version in cold storage.
 *
 * @custom:oz-upgrades-unsafe-allow state-variable-immutable
 */
abstract contract EIP712 is IERC5267 {
    using ShortStrings for *;
    bytes32 private constant TYPE_HASH =
        keccak256("EIP712Domain(string name,string version,uint256 chainId,address verifyingContract)");
    // Cache the domain separator as an immutable value, but also store the chain id that it corresponds to, in order to
    // invalidate the cached domain separator if the chain id changes.
    bytes32 private immutable _cachedDomainSeparator;
    uint256 private immutable _cachedChainId;
    address private immutable _cachedThis;
    bytes32 private immutable _hashedName;
    bytes32 private immutable _hashedVersion;
    ShortString private immutable _name;
    ShortString private immutable _version;
    string private _nameFallback;
    string private _versionFallback;
    /**
     * @dev Initializes the domain separator and parameter caches.
     *
     * The meaning of `name` and `version` is specified in
```

```
    * https://eips.ethereum.org/EIPS/eip-712#definition-of-domainseparator[EIP-712]:
    *
    * - `name`: the user readable name of the signing domain, i.e. the name of the DApp or the
protocol.
    * - `version`: the current major version of the signing domain.
    *
    * NOTE: These parameters cannot be changed except through a
xref:learn::upgrading-smart-contracts.adoc[smart
    * contract upgrade].
    */
   constructor(string memory name, string memory version) {
      _name = name.toShortStringWithFallback(_nameFallback);
      _version = version.toShortStringWithFallback(_versionFallback);
      _hashedName = keccak256(bytes(name));
      _hashedVersion = keccak256(bytes(version));
      _cachedChainId = block.chainid;
      _cachedDomainSeparator = _buildDomainSeparator();
      _cachedThis = address(this);
   }
   /**
    * @dev Returns the domain separator for the current chain.
    */
   function _domainSeparatorV4() internal view returns (bytes32) {
      if (address(this) == _cachedThis && block.chainid == _cachedChainId) {
         return _cachedDomainSeparator;
      } else {
         return _buildDomainSeparator();
      }
   }
   function _buildDomainSeparator() private view returns (bytes32) {
      return keccak256(abi.encode(TYPE_HASH, _hashedName, _hashedVersion,
block.chainid, address(this)));
   }
   /**
    * @dev Given an already
https://eips.ethereum.org/EIPS/eip-712#definition-of-hashstruct[hashed struct], this
    * function returns the hash of the fully encoded EIP712 message for this domain.
    *
    * This hash can be used together with {ECDSA-recover} to obtain the signer of a message.
For example:
    *
    * ```solidity
    * bytes32 digest = _hashTypedDataV4(keccak256(abi.encode(
    *     keccak256("Mail(address to,string contents)"),
```

```
 *     mailTo,
 *     keccak256(bytes(mailContents))
 * )));
 * address signer = ECDSA.recover(digest, signature);
 * ```
 */
function _hashTypedDataV4(bytes32 structHash) internal view virtual returns (bytes32) {
    return MessageHashUtils.toTypedDataHash(_domainSeparatorV4(), structHash);
}
/**
 * @dev See {IERC-5267}.
 */
function eip712Domain()
    public
    view
    virtual
    returns (
        bytes1 fields,
        string memory name,
        string memory version,
        uint256 chainId,
        address verifyingContract,
        bytes32 salt,
        uint256[] memory extensions
    )
{
    return (
        hex"0f", // 01111
        _EIP712Name(),
        _EIP712Version(),
        block.chainid,
        address(this),
        bytes32(0),
        new uint256[](0)
    );
}
/**
 * @dev The name parameter for the EIP712 domain.
 *
 * NOTE: By default this function reads _name which is an immutable value.
 * It only reads from storage if necessary (in case the value is too large to fit in a ShortString).
 */
// solhint-disable-next-line func-name-mixedcase
function _EIP712Name() internal view returns (string memory) {
```

```
        return _name.toStringWithFallback(_nameFallback);
    }
    /**
     * @dev The version parameter for the EIP712 domain.
     *
     * NOTE: By default this function reads _version which is an immutable value.
     * It only reads from storage if necessary (in case the value is too large to fit in a ShortString).
     */
    // solhint-disable-next-line func-name-mixedcase
    function _EIP712Version() internal view returns (string memory) {
        return _version.toStringWithFallback(_versionFallback);
    }
}


// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts (last updated v5.0.0) (interfaces/IERC5805.sol)
pragma solidity ^0.8.20;
import {IVotes} from "../governance/utils/IVotes.sol";
import {IERC6372} from "./IERC6372.sol";
interface IERC5805 is IERC6372, IVotes {}


// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts (last updated v5.1.0) (utils/math/SafeCast.sol)
// This file was procedurally generated from scripts/generate/templates/SafeCast.js.
pragma solidity ^0.8.20;
/**
 * @dev Wrappers over Solidity's uintXX/intXX/bool casting operators with added overflow
 * checks.
 *
 * Downcasting from uint256/int256 in Solidity does not revert on overflow. This can
 * easily result in undesired exploitation or bugs, since developers usually
 * assume that overflows raise errors. `SafeCast` restores this intuition by
 * reverting the transaction when such an operation overflows.
 *
 * Using this library instead of the unchecked operations eliminates an entire
 * class of bugs, so it's recommended to use it always.
 */
library SafeCast {
    /**
     * @dev Value doesn't fit in an uint of `bits` size.
     */
    error SafeCastOverflowedUintDowncast(uint8 bits, uint256 value);
```

```
/**
 * @dev An int value doesn't fit in an uint of `bits` size.
 */
error SafeCastOverflowedIntToUint(int256 value);
/**
 * @dev Value doesn't fit in an int of `bits` size.
 */
error SafeCastOverflowedIntDowncast(uint8 bits, int256 value);
/**
 * @dev An uint value doesn't fit in an int of `bits` size.
 */
error SafeCastOverflowedUintToInt(uint256 value);
/**
 * @dev Returns the downcasted uint248 from uint256, reverting on
 * overflow (when the input is greater than largest uint248).
 *
 * Counterpart to Solidity's `uint248` operator.
 *
 * Requirements:
 *
 * - input must fit into 248 bits
 */
function toUint248(uint256 value) internal pure returns (uint248) {
    if (value > type(uint248).max) {
        revert SafeCastOverflowedUintDowncast(248, value);
    }
    return uint248(value);
}
/**
 * @dev Returns the downcasted uint240 from uint256, reverting on
 * overflow (when the input is greater than largest uint240).
 *
 * Counterpart to Solidity's `uint240` operator.
 *
 * Requirements:
 *
 * - input must fit into 240 bits
 */
function toUint240(uint256 value) internal pure returns (uint240) {
    if (value > type(uint240).max) {
        revert SafeCastOverflowedUintDowncast(240, value);
    }
    return uint240(value);
}
```

```
/**
 * @dev Returns the downcasted uint232 from uint256, reverting on
 * overflow (when the input is greater than largest uint232).
 *
 * Counterpart to Solidity's `uint232` operator.
 *
 * Requirements:
 *
 * - input must fit into 232 bits
 */
function toUint232(uint256 value) internal pure returns (uint232) {
    if (value > type(uint232).max) {
        revert SafeCastOverflowedUintDowncast(232, value);
    }
    return uint232(value);
}
/**
 * @dev Returns the downcasted uint224 from uint256, reverting on
 * overflow (when the input is greater than largest uint224).
 *
 * Counterpart to Solidity's `uint224` operator.
 *
 * Requirements:
 *
 * - input must fit into 224 bits
 */
function toUint224(uint256 value) internal pure returns (uint224) {
    if (value > type(uint224).max) {
        revert SafeCastOverflowedUintDowncast(224, value);
    }
    return uint224(value);
}
/**
 * @dev Returns the downcasted uint216 from uint256, reverting on
 * overflow (when the input is greater than largest uint216).
 *
 * Counterpart to Solidity's `uint216` operator.
 *
 * Requirements:
 *
 * - input must fit into 216 bits
 */
function toUint216(uint256 value) internal pure returns (uint216) {
    if (value > type(uint216).max) {
```

```
        revert SafeCastOverflowedUintDowncast(216, value);
    }
    return uint216(value);
}
/**
 * @dev Returns the downcasted uint208 from uint256, reverting on
 * overflow (when the input is greater than largest uint208).
 *
 * Counterpart to Solidity's `uint208` operator.
 *
 * Requirements:
 *
 * - input must fit into 208 bits
 */
function toUint208(uint256 value) internal pure returns (uint208) {
    if (value > type(uint208).max) {
        revert SafeCastOverflowedUintDowncast(208, value);
    }
    return uint208(value);
}
/**
 * @dev Returns the downcasted uint200 from uint256, reverting on
 * overflow (when the input is greater than largest uint200).
 *
 * Counterpart to Solidity's `uint200` operator.
 *
 * Requirements:
 *
 * - input must fit into 200 bits
 */
function toUint200(uint256 value) internal pure returns (uint200) {
    if (value > type(uint200).max) {
        revert SafeCastOverflowedUintDowncast(200, value);
    }
    return uint200(value);
}
/**
 * @dev Returns the downcasted uint192 from uint256, reverting on
 * overflow (when the input is greater than largest uint192).
 *
 * Counterpart to Solidity's `uint192` operator.
 *
 * Requirements:
 *
```

```
     * - input must fit into 192 bits
     */
    function toUint192(uint256 value) internal pure returns (uint192) {
        if (value > type(uint192).max) {
            revert SafeCastOverflowedUintDowncast(192, value);
        }
        return uint192(value);
    }
    /**
     * @dev Returns the downcasted uint184 from uint256, reverting on
     * overflow (when the input is greater than largest uint184).
     *
     * Counterpart to Solidity's `uint184` operator.
     *
     * Requirements:
     *
     * - input must fit into 184 bits
     */
    function toUint184(uint256 value) internal pure returns (uint184) {
        if (value > type(uint184).max) {
            revert SafeCastOverflowedUintDowncast(184, value);
        }
        return uint184(value);
    }
    /**
     * @dev Returns the downcasted uint176 from uint256, reverting on
     * overflow (when the input is greater than largest uint176).
     *
     * Counterpart to Solidity's `uint176` operator.
     *
     * Requirements:
     *
     * - input must fit into 176 bits
     */
    function toUint176(uint256 value) internal pure returns (uint176) {
        if (value > type(uint176).max) {
            revert SafeCastOverflowedUintDowncast(176, value);
        }
        return uint176(value);
    }
    /**
     * @dev Returns the downcasted uint168 from uint256, reverting on
     * overflow (when the input is greater than largest uint168).
     *
```

```
     * Counterpart to Solidity's `uint168` operator.
     *
     * Requirements:
     *
     * - input must fit into 168 bits
     */
    function toUint168(uint256 value) internal pure returns (uint168) {
        if (value > type(uint168).max) {
            revert SafeCastOverflowedUintDowncast(168, value);
        }
        return uint168(value);
    }
    /**
     * @dev Returns the downcasted uint160 from uint256, reverting on
     * overflow (when the input is greater than largest uint160).
     *
     * Counterpart to Solidity's `uint160` operator.
     *
     * Requirements:
     *
     * - input must fit into 160 bits
     */
    function toUint160(uint256 value) internal pure returns (uint160) {
        if (value > type(uint160).max) {
            revert SafeCastOverflowedUintDowncast(160, value);
        }
        return uint160(value);
    }
    /**
     * @dev Returns the downcasted uint152 from uint256, reverting on
     * overflow (when the input is greater than largest uint152).
     *
     * Counterpart to Solidity's `uint152` operator.
     *
     * Requirements:
     *
     * - input must fit into 152 bits
     */
    function toUint152(uint256 value) internal pure returns (uint152) {
        if (value > type(uint152).max) {
            revert SafeCastOverflowedUintDowncast(152, value);
        }
        return uint152(value);
    }
```

```
/**
 * @dev Returns the downcasted uint144 from uint256, reverting on
 * overflow (when the input is greater than largest uint144).
 *
 * Counterpart to Solidity's `uint144` operator.
 *
 * Requirements:
 *
 * - input must fit into 144 bits
 */
function toUint144(uint256 value) internal pure returns (uint144) {
    if (value > type(uint144).max) {
        revert SafeCastOverflowedUintDowncast(144, value);
    }
    return uint144(value);
}
/**
 * @dev Returns the downcasted uint136 from uint256, reverting on
 * overflow (when the input is greater than largest uint136).
 *
 * Counterpart to Solidity's `uint136` operator.
 *
 * Requirements:
 *
 * - input must fit into 136 bits
 */
function toUint136(uint256 value) internal pure returns (uint136) {
    if (value > type(uint136).max) {
        revert SafeCastOverflowedUintDowncast(136, value);
    }
    return uint136(value);
}
/**
 * @dev Returns the downcasted uint128 from uint256, reverting on
 * overflow (when the input is greater than largest uint128).
 *
 * Counterpart to Solidity's `uint128` operator.
 *
 * Requirements:
 *
 * - input must fit into 128 bits
 */
function toUint128(uint256 value) internal pure returns (uint128) {
    if (value > type(uint128).max) {
```

```
        revert SafeCastOverflowedUintDowncast(128, value);
    }
    return uint128(value);
}
/**
 * @dev Returns the downcasted uint120 from uint256, reverting on
 * overflow (when the input is greater than largest uint120).
 *
 * Counterpart to Solidity's `uint120` operator.
 *
 * Requirements:
 *
 * - input must fit into 120 bits
 */
function toUint120(uint256 value) internal pure returns (uint120) {
    if (value > type(uint120).max) {
        revert SafeCastOverflowedUintDowncast(120, value);
    }
    return uint120(value);
}
/**
 * @dev Returns the downcasted uint112 from uint256, reverting on
 * overflow (when the input is greater than largest uint112).
 *
 * Counterpart to Solidity's `uint112` operator.
 *
 * Requirements:
 *
 * - input must fit into 112 bits
 */
function toUint112(uint256 value) internal pure returns (uint112) {
    if (value > type(uint112).max) {
        revert SafeCastOverflowedUintDowncast(112, value);
    }
    return uint112(value);
}
/**
 * @dev Returns the downcasted uint104 from uint256, reverting on
 * overflow (when the input is greater than largest uint104).
 *
 * Counterpart to Solidity's `uint104` operator.
 *
 * Requirements:
 *
```

```
     * - input must fit into 104 bits
     */
    function toUint104(uint256 value) internal pure returns (uint104) {
        if (value > type(uint104).max) {
            revert SafeCastOverflowedUintDowncast(104, value);
        }
        return uint104(value);
    }
    /**
     * @dev Returns the downcasted uint96 from uint256, reverting on
     * overflow (when the input is greater than largest uint96).
     *
     * Counterpart to Solidity's `uint96` operator.
     *
     * Requirements:
     *
     * - input must fit into 96 bits
     */
    function toUint96(uint256 value) internal pure returns (uint96) {
        if (value > type(uint96).max) {
            revert SafeCastOverflowedUintDowncast(96, value);
        }
        return uint96(value);
    }
    /**
     * @dev Returns the downcasted uint88 from uint256, reverting on
     * overflow (when the input is greater than largest uint88).
     *
     * Counterpart to Solidity's `uint88` operator.
     *
     * Requirements:
     *
     * - input must fit into 88 bits
     */
    function toUint88(uint256 value) internal pure returns (uint88) {
        if (value > type(uint88).max) {
            revert SafeCastOverflowedUintDowncast(88, value);
        }
        return uint88(value);
    }
    /**
     * @dev Returns the downcasted uint80 from uint256, reverting on
     * overflow (when the input is greater than largest uint80).
     *
```

```
     * Counterpart to Solidity's `uint80` operator.
     *
     * Requirements:
     *
     * - input must fit into 80 bits
     */
    function toUint80(uint256 value) internal pure returns (uint80) {
        if (value > type(uint80).max) {
            revert SafeCastOverflowedUintDowncast(80, value);
        }
        return uint80(value);
    }
    /**
     * @dev Returns the downcasted uint72 from uint256, reverting on
     * overflow (when the input is greater than largest uint72).
     *
     * Counterpart to Solidity's `uint72` operator.
     *
     * Requirements:
     *
     * - input must fit into 72 bits
     */
    function toUint72(uint256 value) internal pure returns (uint72) {
        if (value > type(uint72).max) {
            revert SafeCastOverflowedUintDowncast(72, value);
        }
        return uint72(value);
    }
    /**
     * @dev Returns the downcasted uint64 from uint256, reverting on
     * overflow (when the input is greater than largest uint64).
     *
     * Counterpart to Solidity's `uint64` operator.
     *
     * Requirements:
     *
     * - input must fit into 64 bits
     */
    function toUint64(uint256 value) internal pure returns (uint64) {
        if (value > type(uint64).max) {
            revert SafeCastOverflowedUintDowncast(64, value);
        }
        return uint64(value);
    }
```

```
/**
 * @dev Returns the downcasted uint56 from uint256, reverting on
 * overflow (when the input is greater than largest uint56).
 *
 * Counterpart to Solidity's `uint56` operator.
 *
 * Requirements:
 *
 * - input must fit into 56 bits
 */
function toUint56(uint256 value) internal pure returns (uint56) {
    if (value > type(uint56).max) {
        revert SafeCastOverflowedUintDowncast(56, value);
    }
    return uint56(value);
}
/**
 * @dev Returns the downcasted uint48 from uint256, reverting on
 * overflow (when the input is greater than largest uint48).
 *
 * Counterpart to Solidity's `uint48` operator.
 *
 * Requirements:
 *
 * - input must fit into 48 bits
 */
function toUint48(uint256 value) internal pure returns (uint48) {
    if (value > type(uint48).max) {
        revert SafeCastOverflowedUintDowncast(48, value);
    }
    return uint48(value);
}
/**
 * @dev Returns the downcasted uint40 from uint256, reverting on
 * overflow (when the input is greater than largest uint40).
 *
 * Counterpart to Solidity's `uint40` operator.
 *
 * Requirements:
 *
 * - input must fit into 40 bits
 */
function toUint40(uint256 value) internal pure returns (uint40) {
    if (value > type(uint40).max) {
```

```
        revert SafeCastOverflowedUintDowncast(40, value);
    }
    return uint40(value);
}
/**
 * @dev Returns the downcasted uint32 from uint256, reverting on
 * overflow (when the input is greater than largest uint32).
 *
 * Counterpart to Solidity's `uint32` operator.
 *
 * Requirements:
 *
 * - input must fit into 32 bits
 */
function toUint32(uint256 value) internal pure returns (uint32) {
    if (value > type(uint32).max) {
        revert SafeCastOverflowedUintDowncast(32, value);
    }
    return uint32(value);
}
/**
 * @dev Returns the downcasted uint24 from uint256, reverting on
 * overflow (when the input is greater than largest uint24).
 *
 * Counterpart to Solidity's `uint24` operator.
 *
 * Requirements:
 *
 * - input must fit into 24 bits
 */
function toUint24(uint256 value) internal pure returns (uint24) {
    if (value > type(uint24).max) {
        revert SafeCastOverflowedUintDowncast(24, value);
    }
    return uint24(value);
}
/**
 * @dev Returns the downcasted uint16 from uint256, reverting on
 * overflow (when the input is greater than largest uint16).
 *
 * Counterpart to Solidity's `uint16` operator.
 *
 * Requirements:
 *
```

```
 * - input must fit into 16 bits
 */
function toUint16(uint256 value) internal pure returns (uint16) {
    if (value > type(uint16).max) {
        revert SafeCastOverflowedUintDowncast(16, value);
    }
    return uint16(value);
}
/**
 * @dev Returns the downcasted uint8 from uint256, reverting on
 * overflow (when the input is greater than largest uint8).
 *
 * Counterpart to Solidity's `uint8` operator.
 *
 * Requirements:
 *
 * - input must fit into 8 bits
 */
function toUint8(uint256 value) internal pure returns (uint8) {
    if (value > type(uint8).max) {
        revert SafeCastOverflowedUintDowncast(8, value);
    }
    return uint8(value);
}
/**
 * @dev Converts a signed int256 into an unsigned uint256.
 *
 * Requirements:
 *
 * - input must be greater than or equal to 0.
 */
function toUint256(int256 value) internal pure returns (uint256) {
    if (value < 0) {
        revert SafeCastOverflowedIntToUint(value);
    }
    return uint256(value);
}
/**
 * @dev Returns the downcasted int248 from int256, reverting on
 * overflow (when the input is less than smallest int248 or
 * greater than largest int248).
 *
 * Counterpart to Solidity's `int248` operator.
 *
```

```
     * Requirements:
     *
     * - input must fit into 248 bits
     */
    function toInt248(int256 value) internal pure returns (int248 downcasted) {
        downcasted = int248(value);
        if (downcasted != value) {
            revert SafeCastOverflowedIntDowncast(248, value);
        }
    }
    /**
     * @dev Returns the downcasted int240 from int256, reverting on
     * overflow (when the input is less than smallest int240 or
     * greater than largest int240).
     *
     * Counterpart to Solidity's `int240` operator.
     *
     * Requirements:
     *
     * - input must fit into 240 bits
     */
    function toInt240(int256 value) internal pure returns (int240 downcasted) {
        downcasted = int240(value);
        if (downcasted != value) {
            revert SafeCastOverflowedIntDowncast(240, value);
        }
    }
    /**
     * @dev Returns the downcasted int232 from int256, reverting on
     * overflow (when the input is less than smallest int232 or
     * greater than largest int232).
     *
     * Counterpart to Solidity's `int232` operator.
     *
     * Requirements:
     *
     * - input must fit into 232 bits
     */
    function toInt232(int256 value) internal pure returns (int232 downcasted) {
        downcasted = int232(value);
        if (downcasted != value) {
            revert SafeCastOverflowedIntDowncast(232, value);
        }
    }
```

```
/**
 * @dev Returns the downcasted int224 from int256, reverting on
 * overflow (when the input is less than smallest int224 or
 * greater than largest int224).
 *
 * Counterpart to Solidity's `int224` operator.
 *
 * Requirements:
 *
 * - input must fit into 224 bits
 */
function toInt224(int256 value) internal pure returns (int224 downcasted) {
    downcasted = int224(value);
    if (downcasted != value) {
        revert SafeCastOverflowedIntDowncast(224, value);
    }
}
/**
 * @dev Returns the downcasted int216 from int256, reverting on
 * overflow (when the input is less than smallest int216 or
 * greater than largest int216).
 *
 * Counterpart to Solidity's `int216` operator.
 *
 * Requirements:
 *
 * - input must fit into 216 bits
 */
function toInt216(int256 value) internal pure returns (int216 downcasted) {
    downcasted = int216(value);
    if (downcasted != value) {
        revert SafeCastOverflowedIntDowncast(216, value);
    }
}
/**
 * @dev Returns the downcasted int208 from int256, reverting on
 * overflow (when the input is less than smallest int208 or
 * greater than largest int208).
 *
 * Counterpart to Solidity's `int208` operator.
 *
 * Requirements:
 *
 * - input must fit into 208 bits
```

```
 */
function toInt208(int256 value) internal pure returns (int208 downcasted) {
    downcasted = int208(value);
    if (downcasted != value) {
        revert SafeCastOverflowedIntDowncast(208, value);
    }
}
/**
 * @dev Returns the downcasted int200 from int256, reverting on
 * overflow (when the input is less than smallest int200 or
 * greater than largest int200).
 *
 * Counterpart to Solidity's `int200` operator.
 *
 * Requirements:
 *
 * - input must fit into 200 bits
 */
function toInt200(int256 value) internal pure returns (int200 downcasted) {
    downcasted = int200(value);
    if (downcasted != value) {
        revert SafeCastOverflowedIntDowncast(200, value);
    }
}
/**
 * @dev Returns the downcasted int192 from int256, reverting on
 * overflow (when the input is less than smallest int192 or
 * greater than largest int192).
 *
 * Counterpart to Solidity's `int192` operator.
 *
 * Requirements:
 *
 * - input must fit into 192 bits
 */
function toInt192(int256 value) internal pure returns (int192 downcasted) {
    downcasted = int192(value);
    if (downcasted != value) {
        revert SafeCastOverflowedIntDowncast(192, value);
    }
}
/**
 * @dev Returns the downcasted int184 from int256, reverting on
 * overflow (when the input is less than smallest int184 or
```

```
     * greater than largest int184).
     *
     * Counterpart to Solidity's `int184` operator.
     *
     * Requirements:
     *
     * - input must fit into 184 bits
     */
    function toInt184(int256 value) internal pure returns (int184 downcasted) {
        downcasted = int184(value);
        if (downcasted != value) {
            revert SafeCastOverflowedIntDowncast(184, value);
        }
    }
    /**
     * @dev Returns the downcasted int176 from int256, reverting on
     * overflow (when the input is less than smallest int176 or
     * greater than largest int176).
     *
     * Counterpart to Solidity's `int176` operator.
     *
     * Requirements:
     *
     * - input must fit into 176 bits
     */
    function toInt176(int256 value) internal pure returns (int176 downcasted) {
        downcasted = int176(value);
        if (downcasted != value) {
            revert SafeCastOverflowedIntDowncast(176, value);
        }
    }
    /**
     * @dev Returns the downcasted int168 from int256, reverting on
     * overflow (when the input is less than smallest int168 or
     * greater than largest int168).
     *
     * Counterpart to Solidity's `int168` operator.
     *
     * Requirements:
     *
     * - input must fit into 168 bits
     */
    function toInt168(int256 value) internal pure returns (int168 downcasted) {
        downcasted = int168(value);
```

```
    if (downcasted != value) {
        revert SafeCastOverflowedIntDowncast(168, value);
    }
}
/**
 * @dev Returns the downcasted int160 from int256, reverting on
 * overflow (when the input is less than smallest int160 or
 * greater than largest int160).
 *
 * Counterpart to Solidity's `int160` operator.
 *
 * Requirements:
 *
 * - input must fit into 160 bits
 */
function toInt160(int256 value) internal pure returns (int160 downcasted) {
    downcasted = int160(value);
    if (downcasted != value) {
        revert SafeCastOverflowedIntDowncast(160, value);
    }
}
/**
 * @dev Returns the downcasted int152 from int256, reverting on
 * overflow (when the input is less than smallest int152 or
 * greater than largest int152).
 *
 * Counterpart to Solidity's `int152` operator.
 *
 * Requirements:
 *
 * - input must fit into 152 bits
 */
function toInt152(int256 value) internal pure returns (int152 downcasted) {
    downcasted = int152(value);
    if (downcasted != value) {
        revert SafeCastOverflowedIntDowncast(152, value);
    }
}
/**
 * @dev Returns the downcasted int144 from int256, reverting on
 * overflow (when the input is less than smallest int144 or
 * greater than largest int144).
 *
 * Counterpart to Solidity's `int144` operator.
```

```
     *
     * Requirements:
     *
     * - input must fit into 144 bits
     */
    function toInt144(int256 value) internal pure returns (int144 downcasted) {
        downcasted = int144(value);
        if (downcasted != value) {
            revert SafeCastOverflowedIntDowncast(144, value);
        }
    }
    /**
     * @dev Returns the downcasted int136 from int256, reverting on
     * overflow (when the input is less than smallest int136 or
     * greater than largest int136).
     *
     * Counterpart to Solidity's `int136` operator.
     *
     * Requirements:
     *
     * - input must fit into 136 bits
     */
    function toInt136(int256 value) internal pure returns (int136 downcasted) {
        downcasted = int136(value);
        if (downcasted != value) {
            revert SafeCastOverflowedIntDowncast(136, value);
        }
    }
    /**
     * @dev Returns the downcasted int128 from int256, reverting on
     * overflow (when the input is less than smallest int128 or
     * greater than largest int128).
     *
     * Counterpart to Solidity's `int128` operator.
     *
     * Requirements:
     *
     * - input must fit into 128 bits
     */
    function toInt128(int256 value) internal pure returns (int128 downcasted) {
        downcasted = int128(value);
        if (downcasted != value) {
            revert SafeCastOverflowedIntDowncast(128, value);
        }
```

```
}
/**
 * @dev Returns the downcasted int120 from int256, reverting on
 * overflow (when the input is less than smallest int120 or
 * greater than largest int120).
 *
 * Counterpart to Solidity's `int120` operator.
 *
 * Requirements:
 *
 * - input must fit into 120 bits
 */
function toInt120(int256 value) internal pure returns (int120 downcasted) {
    downcasted = int120(value);
    if (downcasted != value) {
        revert SafeCastOverflowedIntDowncast(120, value);
    }
}
/**
 * @dev Returns the downcasted int112 from int256, reverting on
 * overflow (when the input is less than smallest int112 or
 * greater than largest int112).
 *
 * Counterpart to Solidity's `int112` operator.
 *
 * Requirements:
 *
 * - input must fit into 112 bits
 */
function toInt112(int256 value) internal pure returns (int112 downcasted) {
    downcasted = int112(value);
    if (downcasted != value) {
        revert SafeCastOverflowedIntDowncast(112, value);
    }
}
/**
 * @dev Returns the downcasted int104 from int256, reverting on
 * overflow (when the input is less than smallest int104 or
 * greater than largest int104).
 *
 * Counterpart to Solidity's `int104` operator.
 *
 * Requirements:
 *
```

```
 * - input must fit into 104 bits
 */
function toInt104(int256 value) internal pure returns (int104 downcasted) {
    downcasted = int104(value);
    if (downcasted != value) {
        revert SafeCastOverflowedIntDowncast(104, value);
    }
}
/**
 * @dev Returns the downcasted int96 from int256, reverting on
 * overflow (when the input is less than smallest int96 or
 * greater than largest int96).
 *
 * Counterpart to Solidity's `int96` operator.
 *
 * Requirements:
 *
 * - input must fit into 96 bits
 */
function toInt96(int256 value) internal pure returns (int96 downcasted) {
    downcasted = int96(value);
    if (downcasted != value) {
        revert SafeCastOverflowedIntDowncast(96, value);
    }
}
/**
 * @dev Returns the downcasted int88 from int256, reverting on
 * overflow (when the input is less than smallest int88 or
 * greater than largest int88).
 *
 * Counterpart to Solidity's `int88` operator.
 *
 * Requirements:
 *
 * - input must fit into 88 bits
 */
function toInt88(int256 value) internal pure returns (int88 downcasted) {
    downcasted = int88(value);
    if (downcasted != value) {
        revert SafeCastOverflowedIntDowncast(88, value);
    }
}
/**
 * @dev Returns the downcasted int80 from int256, reverting on
```

```
 * overflow (when the input is less than smallest int80 or
 * greater than largest int80).
 *
 * Counterpart to Solidity's `int80` operator.
 *
 * Requirements:
 *
 * - input must fit into 80 bits
 */
function toInt80(int256 value) internal pure returns (int80 downcasted) {
    downcasted = int80(value);
    if (downcasted != value) {
        revert SafeCastOverflowedIntDowncast(80, value);
    }
}
/**
 * @dev Returns the downcasted int72 from int256, reverting on
 * overflow (when the input is less than smallest int72 or
 * greater than largest int72).
 *
 * Counterpart to Solidity's `int72` operator.
 *
 * Requirements:
 *
 * - input must fit into 72 bits
 */
function toInt72(int256 value) internal pure returns (int72 downcasted) {
    downcasted = int72(value);
    if (downcasted != value) {
        revert SafeCastOverflowedIntDowncast(72, value);
    }
}
/**
 * @dev Returns the downcasted int64 from int256, reverting on
 * overflow (when the input is less than smallest int64 or
 * greater than largest int64).
 *
 * Counterpart to Solidity's `int64` operator.
 *
 * Requirements:
 *
 * - input must fit into 64 bits
 */
function toInt64(int256 value) internal pure returns (int64 downcasted) {
```

```
        downcasted = int64(value);
        if (downcasted != value) {
            revert SafeCastOverflowedIntDowncast(64, value);
        }
    }
    /**
     * @dev Returns the downcasted int56 from int256, reverting on
     * overflow (when the input is less than smallest int56 or
     * greater than largest int56).
     *
     * Counterpart to Solidity's `int56` operator.
     *
     * Requirements:
     *
     * - input must fit into 56 bits
     */
    function toInt56(int256 value) internal pure returns (int56 downcasted) {
        downcasted = int56(value);
        if (downcasted != value) {
            revert SafeCastOverflowedIntDowncast(56, value);
        }
    }
    /**
     * @dev Returns the downcasted int48 from int256, reverting on
     * overflow (when the input is less than smallest int48 or
     * greater than largest int48).
     *
     * Counterpart to Solidity's `int48` operator.
     *
     * Requirements:
     *
     * - input must fit into 48 bits
     */
    function toInt48(int256 value) internal pure returns (int48 downcasted) {
        downcasted = int48(value);
        if (downcasted != value) {
            revert SafeCastOverflowedIntDowncast(48, value);
        }
    }
    /**
     * @dev Returns the downcasted int40 from int256, reverting on
     * overflow (when the input is less than smallest int40 or
     * greater than largest int40).
     *
```

```
 * Counterpart to Solidity's `int40` operator.
 *
 * Requirements:
 *
 * - input must fit into 40 bits
 */
function toInt40(int256 value) internal pure returns (int40 downcasted) {
    downcasted = int40(value);
    if (downcasted != value) {
        revert SafeCastOverflowedIntDowncast(40, value);
    }
}
/**
 * @dev Returns the downcasted int32 from int256, reverting on
 * overflow (when the input is less than smallest int32 or
 * greater than largest int32).
 *
 * Counterpart to Solidity's `int32` operator.
 *
 * Requirements:
 *
 * - input must fit into 32 bits
 */
function toInt32(int256 value) internal pure returns (int32 downcasted) {
    downcasted = int32(value);
    if (downcasted != value) {
        revert SafeCastOverflowedIntDowncast(32, value);
    }
}
/**
 * @dev Returns the downcasted int24 from int256, reverting on
 * overflow (when the input is less than smallest int24 or
 * greater than largest int24).
 *
 * Counterpart to Solidity's `int24` operator.
 *
 * Requirements:
 *
 * - input must fit into 24 bits
 */
function toInt24(int256 value) internal pure returns (int24 downcasted) {
    downcasted = int24(value);
    if (downcasted != value) {
        revert SafeCastOverflowedIntDowncast(24, value);
```

```
    }
}
/**
 * @dev Returns the downcasted int16 from int256, reverting on
 * overflow (when the input is less than smallest int16 or
 * greater than largest int16).
 *
 * Counterpart to Solidity's `int16` operator.
 *
 * Requirements:
 *
 * - input must fit into 16 bits
 */
function toInt16(int256 value) internal pure returns (int16 downcasted) {
    downcasted = int16(value);
    if (downcasted != value) {
        revert SafeCastOverflowedIntDowncast(16, value);
    }
}
/**
 * @dev Returns the downcasted int8 from int256, reverting on
 * overflow (when the input is less than smallest int8 or
 * greater than largest int8).
 *
 * Counterpart to Solidity's `int8` operator.
 *
 * Requirements:
 *
 * - input must fit into 8 bits
 */
function toInt8(int256 value) internal pure returns (int8 downcasted) {
    downcasted = int8(value);
    if (downcasted != value) {
        revert SafeCastOverflowedIntDowncast(8, value);
    }
}
/**
 * @dev Converts an unsigned uint256 into a signed int256.
 *
 * Requirements:
 *
 * - input must be less than or equal to maxInt256.
 */
function toInt256(uint256 value) internal pure returns (int256) {
```

```
        // Note: Unsafe cast below is okay because `type(int256).max` is guaranteed to be positive
        if (value > uint256(type(int256).max)) {
            revert SafeCastOverflowedUintToInt(value);
        }
        return int256(value);
    }
    /**
     * @dev Cast a boolean (false or true) to a uint256 (0 or 1) with no jump.
     */
    function toUint(bool b) internal pure returns (uint256 u) {
        assembly ("memory-safe") {
            u := iszero(iszero(b))
        }
    }
}


// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts (last updated v5.1.0) (utils/types/Time.sol)
pragma solidity ^0.8.20;
import {Math} from "../math/Math.sol";
import {SafeCast} from "../math/SafeCast.sol";
/**
 * @dev This library provides helpers for manipulating time-related objects.
 *
 * It uses the following types:
 * - `uint48` for timepoints
 * - `uint32` for durations
 *
 * While the library doesn't provide specific types for timepoints and duration, it does provide:
 * - a `Delay` type to represent duration that can be programmed to change value automatically
at a given point
 * - additional helper functions
 */
library Time {
    using Time for *;
    /**
     * @dev Get the block timestamp as a Timepoint.
     */
    function timestamp() internal view returns (uint48) {
        return SafeCast.toUint48(block.timestamp);
    }
    /**
     * @dev Get the block number as a Timepoint.
```

```
     */
    function blockNumber() internal view returns (uint48) {
        return SafeCast.toUint48(block.number);
    }
    // ================================================== Delay ==================================================
    /**
     * @dev A `Delay` is a uint32 duration that can be programmed to change value automatically at a given point in the
     * future. The "effect" timepoint describes when the transitions happens from the "old" value to the "new" value.
     * This allows updating the delay applied to some operation while keeping some guarantees.
     *
     * In particular, the {update} function guarantees that if the delay is reduced, the old delay still applies for
     * some time. For example if the delay is currently 7 days to do an upgrade, the admin should not be able to set
     * the delay to 0 and upgrade immediately. If the admin wants to reduce the delay, the old delay (7 days) should
     * still apply for some time.
     *
     *
     * The `Delay` type is 112 bits long, and packs the following:
     *
     * ```
     *   | [uint48]: effect date (timepoint)
     *   |           | [uint32]: value before (duration)
     *   ↓           ↓       ↓ [uint32]: value after (duration)
     * 0xAAAAAAAAAAAABBBBBBBBCCCCCCCC
     * ```
     *
     * NOTE: The {get} and {withUpdate} functions operate using timestamps. Block number based delays are not currently
     * supported.
     */
    type Delay is uint112;
    /**
     * @dev Wrap a duration into a Delay to add the one-step "update in the future" feature
     */
    function toDelay(uint32 duration) internal pure returns (Delay) {
        return Delay.wrap(duration);
    }
    /**
```

```
     * @dev Get the value at a given timepoint plus the pending value and effect timepoint if there
is a scheduled
     * change after this timepoint. If the effect timepoint is 0, then the pending value should not be
considered.
     */
    function _getFullAt(
        Delay self,
        uint48 timepoint
    ) private pure returns (uint32 valueBefore, uint32 valueAfter, uint48 effect) {
        (valueBefore, valueAfter, effect) = self.unpack();
        return effect <= timepoint ? (valueAfter, 0, 0) : (valueBefore, valueAfter, effect);
    }
    /**
     * @dev Get the current value plus the pending value and effect timepoint if there is a
scheduled change. If the
     * effect timepoint is 0, then the pending value should not be considered.
     */
    function getFull(Delay self) internal view returns (uint32 valueBefore, uint32 valueAfter, uint48
effect) {
        return _getFullAt(self, timestamp());
    }
    /**
     * @dev Get the current value.
     */
    function get(Delay self) internal view returns (uint32) {
        (uint32 delay, , ) = self.getFull();
        return delay;
    }
    /**
     * @dev Update a Delay object so that it takes a new duration after a timepoint that is
automatically computed to
     * enforce the old delay at the moment of the update. Returns the updated Delay object and
the timestamp when the
     * new delay becomes effective.
     */
    function withUpdate(
        Delay self,
        uint32 newValue,
        uint32 minSetback
    ) internal view returns (Delay updatedDelay, uint48 effect) {
        uint32 value = self.get();
        uint32 setback = uint32(Math.max(minSetback, value > newValue ? value - newValue : 0));
        effect = timestamp() + setback;
        return (pack(value, newValue, effect), effect);
```

```
    }
    /**
     * @dev Split a delay into its components: valueBefore, valueAfter and effect (transition
timepoint).
     */
    function unpack(Delay self) internal pure returns (uint32 valueBefore, uint32 valueAfter,
uint48 effect) {
        uint112 raw = Delay.unwrap(self);
        valueAfter = uint32(raw);
        valueBefore = uint32(raw >> 32);
        effect = uint48(raw >> 64);
        return (valueBefore, valueAfter, effect);
    }
    /**
     * @dev pack the components into a Delay object.
     */
    function pack(uint32 valueBefore, uint32 valueAfter, uint48 effect) internal pure returns
(Delay) {
        return Delay.wrap((uint112(effect) << 64) | (uint112(valueBefore) << 32) |
uint112(valueAfter));
    }
}


// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts (last updated v5.1.0) (utils/math/Math.sol)
pragma solidity ^0.8.20;
import {Panic} from "../Panic.sol";
import {SafeCast} from "./SafeCast.sol";
/**
 * @dev Standard math utilities missing in the Solidity language.
 */
library Math {
    enum Rounding {
        Floor, // Toward negative infinity
        Ceil, // Toward positive infinity
        Trunc, // Toward zero
        Expand // Away from zero
    }
    /**
     * @dev Returns the addition of two unsigned integers, with an success flag (no overflow).
     */
    function tryAdd(uint256 a, uint256 b) internal pure returns (bool success, uint256 result) {
        unchecked {
```

```
            uint256 c = a + b;
            if (c < a) return (false, 0);
            return (true, c);
        }
    }
    /**
     * @dev Returns the subtraction of two unsigned integers, with an success flag (no overflow).
     */
    function trySub(uint256 a, uint256 b) internal pure returns (bool success, uint256 result) {
        unchecked {
            if (b > a) return (false, 0);
            return (true, a - b);
        }
    }
    /**
     * @dev Returns the multiplication of two unsigned integers, with an success flag (no
overflow).
     */
    function tryMul(uint256 a, uint256 b) internal pure returns (bool success, uint256 result) {
        unchecked {
            // Gas optimization: this is cheaper than requiring 'a' not being zero, but the
            // benefit is lost if 'b' is also tested.
            // See: https://github.com/OpenZeppelin/openzeppelin-contracts/pull/522
            if (a == 0) return (true, 0);
            uint256 c = a * b;
            if (c / a != b) return (false, 0);
            return (true, c);
        }
    }
    /**
     * @dev Returns the division of two unsigned integers, with a success flag (no division by
zero).
     */
    function tryDiv(uint256 a, uint256 b) internal pure returns (bool success, uint256 result) {
        unchecked {
            if (b == 0) return (false, 0);
            return (true, a / b);
        }
    }
    /**
     * @dev Returns the remainder of dividing two unsigned integers, with a success flag (no
division by zero).
     */
    function tryMod(uint256 a, uint256 b) internal pure returns (bool success, uint256 result) {
```

```
        unchecked {
            if (b == 0) return (false, 0);
            return (true, a % b);
        }
    }
    /**
     * @dev Branchless ternary evaluation for `a ? b : c`. Gas costs are constant.
     *
     * IMPORTANT: This function may reduce bytecode size and consume less gas when used standalone.
     * However, the compiler may optimize Solidity ternary operations (i.e. `a ? b : c`) to only compute
     * one branch when needed, making this function more expensive.
     */
    function ternary(bool condition, uint256 a, uint256 b) internal pure returns (uint256) {
        unchecked {
            // branchless ternary works because:
            // b ^ (a ^ b) == a
            // b ^ 0 == b
            return b ^ ((a ^ b) * SafeCast.toUint(condition));
        }
    }
    /**
     * @dev Returns the largest of two numbers.
     */
    function max(uint256 a, uint256 b) internal pure returns (uint256) {
        return ternary(a > b, a, b);
    }
    /**
     * @dev Returns the smallest of two numbers.
     */
    function min(uint256 a, uint256 b) internal pure returns (uint256) {
        return ternary(a < b, a, b);
    }
    /**
     * @dev Returns the average of two numbers. The result is rounded towards
     * zero.
     */
    function average(uint256 a, uint256 b) internal pure returns (uint256) {
        // (a + b) / 2 can overflow.
        return (a & b) + (a ^ b) / 2;
    }
    /**
     * @dev Returns the ceiling of the division of two numbers.
```

```
     *
     * This differs from standard division with `/` in that it rounds towards infinity instead
     * of rounding towards zero.
     */
    function ceilDiv(uint256 a, uint256 b) internal pure returns (uint256) {
        if (b == 0) {
            // Guarantee the same behavior as in a regular Solidity division.
            Panic.panic(Panic.DIVISION_BY_ZERO);
        }
        // The following calculation ensures accurate ceiling division without overflow.
        // Since a is non-zero, (a - 1) / b will not overflow.
        // The largest possible result occurs when (a - 1) / b is type(uint256).max,
        // but the largest value we can obtain is type(uint256).max - 1, which happens
        // when a = type(uint256).max and b = 1.
        unchecked {
            return SafeCast.toUint(a > 0) * ((a - 1) / b + 1);
        }
    }
    /**
     * @dev Calculates floor(x * y / denominator) with full precision. Throws if result overflows a
uint256 or
     * denominator == 0.
     *
     * Original credit to Remco Bloemen under MIT license (https://xn--2-umb.com/21/muldiv) with
further edits by
     * Uniswap Labs also under MIT license.
     */
    function mulDiv(uint256 x, uint256 y, uint256 denominator) internal pure returns (uint256
result) {
        unchecked {
            // 512-bit multiply [prod1 prod0] = x * y. Compute the product mod 2^256 and mod 2^256 - 1,
then use
            // the Chinese Remainder Theorem to reconstruct the 512 bit result. The result is stored
in two 256
            // variables such that product = prod1 * 2^256 + prod0.
            uint256 prod0 = x * y; // Least significant 256 bits of the product
            uint256 prod1; // Most significant 256 bits of the product
            assembly {
                let mm := mulmod(x, y, not(0))
                prod1 := sub(sub(mm, prod0), lt(mm, prod0))
            }
            // Handle non-overflow cases, 256 by 256 division.
            if (prod1 == 0) {
                // Solidity will revert if denominator == 0, unlike the div opcode on its own.
```

```
            // The surrounding unchecked block does not change this fact.
            // See
https://docs.soliditylang.org/en/latest/control-structures.html#checked-or-unchecked-arithmetic.
            return prod0 / denominator;
        }
        // Make sure the result is less than 2²⁵⁶. Also prevents denominator == 0.
        if (denominator <= prod1) {
            Panic.panic(ternary(denominator == 0, Panic.DIVISION_BY_ZERO,
Panic.UNDER_OVERFLOW));
        }
        ///////////////////////////////////////////////
        // 512 by 256 division.
        ///////////////////////////////////////////////
        // Make division exact by subtracting the remainder from [prod1 prod0].
        uint256 remainder;
        assembly {
            // Compute remainder using mulmod.
            remainder := mulmod(x, y, denominator)
            // Subtract 256 bit number from 512 bit number.
            prod1 := sub(prod1, gt(remainder, prod0))
            prod0 := sub(prod0, remainder)
        }
        // Factor powers of two out of denominator and compute largest power of two divisor of
denominator.
        // Always >= 1. See https://cs.stackexchange.com/q/138556/92363.
        uint256 twos = denominator & (0 - denominator);
        assembly {
            // Divide denominator by twos.
            denominator := div(denominator, twos)
            // Divide [prod1 prod0] by twos.
            prod0 := div(prod0, twos)
            // Flip twos such that it is 2²⁵⁶ / twos. If twos is zero, then it becomes one.
            twos := add(div(sub(0, twos), twos), 1)
        }
        // Shift in bits from prod1 into prod0.
        prod0 |= prod1 * twos;
        // Invert denominator mod 2²⁵⁶. Now that denominator is an odd number, it has an
inverse modulo 2²⁵⁶ such
        // that denominator * inv ≡ 1 mod 2²⁵⁶. Compute the inverse by starting with a seed that is
correct for
        // four bits. That is, denominator * inv ≡ 1 mod 2⁴.
        uint256 inverse = (3 * denominator) ^ 2;
        // Use the Newton-Raphson iteration to improve the precision. Thanks to Hensel's lifting
lemma, this also
```

```
            // works in modular arithmetic, doubling the correct bits in each step.
            inverse *= 2 - denominator * inverse; // inverse mod 2^8
            inverse *= 2 - denominator * inverse; // inverse mod 2^16
            inverse *= 2 - denominator * inverse; // inverse mod 2^32
            inverse *= 2 - denominator * inverse; // inverse mod 2^64
            inverse *= 2 - denominator * inverse; // inverse mod 2^128
            inverse *= 2 - denominator * inverse; // inverse mod 2^256
            // Because the division is now exact we can divide by multiplying with the modular
inverse of denominator.
            // This will give us the correct result modulo 2^256. Since the preconditions guarantee that
the outcome is
            // less than 2^256, this is the final result. We don't need to compute the high bits of the
result and prod1
            // is no longer required.
            result = prod0 * inverse;
            return result;
        }
    }
    /**
     * @dev Calculates x * y / denominator with full precision, following the selected rounding
direction.
     */
    function mulDiv(uint256 x, uint256 y, uint256 denominator, Rounding rounding) internal pure
returns (uint256) {
        return mulDiv(x, y, denominator) + SafeCast.toUint(unsignedRoundsUp(rounding) &&
mulmod(x, y, denominator) > 0);
    }
    /**
     * @dev Calculate the modular multiplicative inverse of a number in Z/nZ.
     *
     * If n is a prime, then Z/nZ is a field. In that case all elements are inversible, except 0.
     * If n is not a prime, then Z/nZ is not a field, and some elements might not be inversible.
     *
     * If the input value is not inversible, 0 is returned.
     *
     * NOTE: If you know for sure that n is (big) a prime, it may be cheaper to use Fermat's little
theorem and get the
     * inverse using `Math.modExp(a, n - 2, n)`. See {invModPrime}.
     */
    function invMod(uint256 a, uint256 n) internal pure returns (uint256) {
        unchecked {
            if (n == 0) return 0;
            // The inverse modulo is calculated using the Extended Euclidean Algorithm (iterative
version)
```

```
        // Used to compute integers x and y such that: ax + ny = gcd(a, n).
        // When the gcd is 1, then the inverse of a modulo n exists and it's x.
        // ax + ny = 1
        // ax = 1 + (-y)n
        // ax ≡ 1 (mod n) # x is the inverse of a modulo n
        // If the remainder is 0 the gcd is n right away.
        uint256 remainder = a % n;
        uint256 gcd = n;
        // Therefore the initial coefficients are:
        // ax + ny = gcd(a, n) = n
        // 0a + 1n = n
        int256 x = 0;
        int256 y = 1;
        while (remainder != 0) {
            uint256 quotient = gcd / remainder;
            (gcd, remainder) = (
                // The old remainder is the next gcd to try.
                remainder,
                // Compute the next remainder.
                // Can't overflow given that (a % gcd) * (gcd // (a % gcd)) <= gcd
                // where gcd is at most n (capped to type(uint256).max)
                gcd - remainder * quotient
            );
            (x, y) = (
                // Increment the coefficient of a.
                y,
                // Decrement the coefficient of n.
                // Can overflow, but the result is casted to uint256 so that the
                // next value of y is "wrapped around" to a value between 0 and n - 1.
                x - y * int256(quotient)
            );
        }
        if (gcd != 1) return 0; // No inverse exists.
        return ternary(x < 0, n - uint256(-x), uint256(x)); // Wrap the result if it's negative.
    }
}
/**
 * @dev Variant of {invMod}. More efficient, but only works if `p` is known to be a prime greater than `2`.
 *
 * From https://en.wikipedia.org/wiki/Fermat%27s_little_theorem[Fermat's little theorem], we know that if p is
 * prime, then `a**(p-1) ≡ 1 mod p`. As a consequence, we have `a * a**(p-2) ≡ 1 mod p`, which means that
```

```
     * `a**(p-2)` is the modular multiplicative inverse of a in Fp.
     *
     * NOTE: this function does NOT check that `p` is a prime greater than `2`.
     */
    function invModPrime(uint256 a, uint256 p) internal view returns (uint256) {
        unchecked {
            return Math.modExp(a, p - 2, p);
        }
    }
    /**
     * @dev Returns the modular exponentiation of the specified base, exponent and modulus (b
** e % m)
     *
     * Requirements:
     * - modulus can't be zero
     * - underlying staticcall to precompile must succeed
     *
     * IMPORTANT: The result is only valid if the underlying call succeeds. When using this
function, make
     * sure the chain you're using it on supports the precompiled contract for modular
exponentiation
     * at address 0x05 as specified in https://eips.ethereum.org/EIPS/eip-198[EIP-198].
Otherwise,
     * the underlying function will succeed given the lack of a revert, but the result may be
incorrectly
     * interpreted as 0.
     */
    function modExp(uint256 b, uint256 e, uint256 m) internal view returns (uint256) {
        (bool success, uint256 result) = tryModExp(b, e, m);
        if (!success) {
            Panic.panic(Panic.DIVISION_BY_ZERO);
        }
        return result;
    }
    /**
     * @dev Returns the modular exponentiation of the specified base, exponent and modulus (b
** e % m).
     * It includes a success flag indicating if the operation succeeded. Operation will be marked
as failed if trying
     * to operate modulo 0 or if the underlying precompile reverted.
     *
     * IMPORTANT: The result is only valid if the success flag is true. When using this function,
make sure the chain
```

```
     * you're using it on supports the precompiled contract for modular exponentiation at address
0x05 as specified in
     * https://eips.ethereum.org/EIPS/eip-198[EIP-198]. Otherwise, the underlying function will
succeed given the lack
     * of a revert, but the result may be incorrectly interpreted as 0.
     */
    function tryModExp(uint256 b, uint256 e, uint256 m) internal view returns (bool success,
uint256 result) {
        if (m == 0) return (false, 0);
        assembly ("memory-safe") {
            let ptr := mload(0x40)
            // | Offset    | Content    | Content (Hex)                                                      |
            // |-----------|------------|--------------------------------------------------------------------|
            // | 0x00:0x1f | size of b  |
0x0000000000000000000000000000000000000000000000000000000000000020 |
            // | 0x20:0x3f | size of e  |
0x0000000000000000000000000000000000000000000000000000000000000020 |
            // | 0x40:0x5f | size of m  |
0x0000000000000000000000000000000000000000000000000000000000000020 |
            // | 0x60:0x7f | value of b | 0x<.....................................................b> |
            // | 0x80:0x9f | value of e | 0x<.....................................................e> |
            // | 0xa0:0xbf | value of m | 0x<.....................................................m> |
            mstore(ptr, 0x20)
            mstore(add(ptr, 0x20), 0x20)
            mstore(add(ptr, 0x40), 0x20)
            mstore(add(ptr, 0x60), b)
            mstore(add(ptr, 0x80), e)
            mstore(add(ptr, 0xa0), m)
            // Given the result < m, it's guaranteed to fit in 32 bytes,
            // so we can use the memory scratch space located at offset 0.
            success := staticcall(gas(), 0x05, ptr, 0xc0, 0x00, 0x20)
            result := mload(0x00)
        }
    }
    /**
     * @dev Variant of {modExp} that supports inputs of arbitrary length.
     */
    function modExp(bytes memory b, bytes memory e, bytes memory m) internal view returns
(bytes memory) {
        (bool success, bytes memory result) = tryModExp(b, e, m);
        if (!success) {
            Panic.panic(Panic.DIVISION_BY_ZERO);
        }
        return result;
```

```
    }
    /**
     * @dev Variant of {tryModExp} that supports inputs of arbitrary length.
     */
    function tryModExp(
        bytes memory b,
        bytes memory e,
        bytes memory m
    ) internal view returns (bool success, bytes memory result) {
        if (_zeroBytes(m)) return (false, new bytes(0));
        uint256 mLen = m.length;
        // Encode call args in result and move the free memory pointer
        result = abi.encodePacked(b.length, e.length, mLen, b, e, m);
        assembly ("memory-safe") {
            let dataPtr := add(result, 0x20)
            // Write result on top of args to avoid allocating extra memory.
            success := staticcall(gas(), 0x05, dataPtr, mload(result), dataPtr, mLen)
            // Overwrite the length.
            // result.length > returndatasize() is guaranteed because returndatasize() == m.length
            mstore(result, mLen)
            // Set the memory pointer after the returned data.
            mstore(0x40, add(dataPtr, mLen))
        }
    }
    /**
     * @dev Returns whether the provided byte array is zero.
     */
    function _zeroBytes(bytes memory byteArray) private pure returns (bool) {
        for (uint256 i = 0; i < byteArray.length; ++i) {
            if (byteArray[i] != 0) {
                return false;
            }
        }
        return true;
    }
    /**
     * @dev Returns the square root of a number. If the number is not a perfect square, the value
is rounded
     * towards zero.
     *
     * This method is based on Newton's method for computing square roots; the algorithm is
restricted to only
     * using integer operations.
     */
```

```
    function sqrt(uint256 a) internal pure returns (uint256) {
        unchecked {
            // Take care of easy edge cases when a == 0 or a == 1
            if (a <= 1) {
                return a;
            }
            // In this function, we use Newton's method to get a root of `f(x) := x² - a`. It involves building a
            // sequence x_n that converges toward sqrt(a). For each iteration x_n, we also define the error between
            // the current value as `ε_n = | x_n - sqrt(a) |`.
            //
            // For our first estimation, we consider `e` the smallest power of 2 which is bigger than the square root
            // of the target. (i.e. `2**(e-1) ≤ sqrt(a) < 2**e`). We know that `e ≤ 128` because `(2¹²⁸)² = 2²⁵⁶` is
            // bigger than any uint256.
            //
            // By noticing that
            // `2**(e-1) ≤ sqrt(a) < 2**e → (2**(e-1))² ≤ a < (2**e)² → 2**(2*e-2) ≤ a < 2**(2*e)`
            // we can deduce that `e - 1` is `log2(a) / 2`. We can thus compute `x_n = 2**(e-1)` using a method similar
            // to the msb function.
            uint256 aa = a;
            uint256 xn = 1;
            if (aa >= (1 << 128)) {
                aa >>= 128;
                xn <<= 64;
            }
            if (aa >= (1 << 64)) {
                aa >>= 64;
                xn <<= 32;
            }
            if (aa >= (1 << 32)) {
                aa >>= 32;
                xn <<= 16;
            }
            if (aa >= (1 << 16)) {
                aa >>= 16;
                xn <<= 8;
            }
            if (aa >= (1 << 8)) {
                aa >>= 8;
                xn <<= 4;
```

```
        }
        if (aa >= (1 << 4)) {
            aa >>= 4;
            xn <<= 2;
        }
        if (aa >= (1 << 2)) {
            xn <<= 1;
        }
        // We now have x_n such that `x_n = 2**(e-1) ≤ sqrt(a) < 2**e = 2 * x_n`. This implies ε_n
≤ 2**(e-1).
        //
        // We can refine our estimation by noticing that the middle of that interval minimizes the
error.
        // If we move x_n to equal 2**(e-1) + 2**(e-2), then we reduce the error to ε_n ≤ 2**(e-2).
        // This is going to be our x_0 (and ε_0)
        xn = (3 * xn) >> 1; // ε_0 := | x_0 - sqrt(a) | ≤ 2**(e-2)
        // From here, Newton's method give us:
        // x_{n+1} = (x_n + a / x_n) / 2
        //
        // One should note that:
        // x_{n+1}² - a = ((x_n + a / x_n) / 2)² - a
        //              = ((x_n² + a) / (2 * x_n))² - a
        //              = (x_n⁴ + 2 * a * x_n² + a²) / (4 * x_n²) - a
        //              = (x_n⁴ + 2 * a * x_n² + a² - 4 * a * x_n²) / (4 * x_n²)
        //              = (x_n⁴ - 2 * a * x_n² + a²) / (4 * x_n²)
        //              = (x_n² - a)² / (2 * x_n)²
        //              = ((x_n² - a) / (2 * x_n))²
        //              ≥ 0
        // Which proves that for all n ≥ 1, sqrt(a) ≤ x_n
        //
        // This gives us the proof of quadratic convergence of the sequence:
        // ε_{n+1} = | x_{n+1} - sqrt(a) |
        //         = | (x_n + a / x_n) / 2 - sqrt(a) |
        //         = | (x_n² + a - 2*x_n*sqrt(a)) / (2 * x_n) |
        //         = | (x_n - sqrt(a))² / (2 * x_n) |
        //         = | ε_n² / (2 * x_n) |
        //         = ε_n² / | (2 * x_n) |
        //
        // For the first iteration, we have a special case where x_0 is known:
        // ε_1 = ε_0² / | (2 * x_0) |
        //     ≤ (2**(e-2))² / (2 * (2**(e-1) + 2**(e-2)))
        //     ≤ 2**(2*e-4) / (3 * 2**(e-1))
        //     ≤ 2**(e-3) / 3
        //     ≤ 2**(e-3-log2(3))
```

```
            //     ≤ 2**(e-4.5)
            //
            // For the following iterations, we use the fact that, 2**(e-1) ≤ sqrt(a) ≤ x_n:
            // ε_{n+1} = ε_n² / | (2 * x_n) |
            //         ≤ (2**(e-k))² / (2 * 2**(e-1))
            //         ≤ 2**(2*e-2*k) / 2**e
            //         ≤ 2**(e-2*k)
            xn = (xn + a / xn) >> 1; // ε_1 := | x_1 - sqrt(a) | ≤ 2**(e-4.5)  -- special case, see above
            xn = (xn + a / xn) >> 1; // ε_2 := | x_2 - sqrt(a) | ≤ 2**(e-9)    -- general case with k = 4.5
            xn = (xn + a / xn) >> 1; // ε_3 := | x_3 - sqrt(a) | ≤ 2**(e-18)   -- general case with k = 9
            xn = (xn + a / xn) >> 1; // ε_4 := | x_4 - sqrt(a) | ≤ 2**(e-36)   -- general case with k = 18
            xn = (xn + a / xn) >> 1; // ε_5 := | x_5 - sqrt(a) | ≤ 2**(e-72)   -- general case with k = 36
            xn = (xn + a / xn) >> 1; // ε_6 := | x_6 - sqrt(a) | ≤ 2**(e-144)  -- general case with k = 72
            // Because e ≤ 128 (as discussed during the first estimation phase), we know have reached a precision
            // ε_6 ≤ 2**(e-144) < 1. Given we're operating on integers, then we can ensure that xn is now either
            // sqrt(a) or sqrt(a) + 1.
            return xn - SafeCast.toUint(xn > a / xn);
        }
    }
    /**
     * @dev Calculates sqrt(a), following the selected rounding direction.
     */
    function sqrt(uint256 a, Rounding rounding) internal pure returns (uint256) {
        unchecked {
            uint256 result = sqrt(a);
            return result + SafeCast.toUint(unsignedRoundsUp(rounding) && result * result < a);
        }
    }
    /**
     * @dev Return the log in base 2 of a positive value rounded towards zero.
     * Returns 0 if given 0.
     */
    function log2(uint256 value) internal pure returns (uint256) {
        uint256 result = 0;
        uint256 exp;
        unchecked {
            exp = 128 * SafeCast.toUint(value > (1 << 128) - 1);
            value >>= exp;
            result += exp;
            exp = 64 * SafeCast.toUint(value > (1 << 64) - 1);
            value >>= exp;
            result += exp;
```

```
            exp = 32 * SafeCast.toUint(value > (1 << 32) - 1);
            value >>= exp;
            result += exp;
            exp = 16 * SafeCast.toUint(value > (1 << 16) - 1);
            value >>= exp;
            result += exp;
            exp = 8 * SafeCast.toUint(value > (1 << 8) - 1);
            value >>= exp;
            result += exp;
            exp = 4 * SafeCast.toUint(value > (1 << 4) - 1);
            value >>= exp;
            result += exp;
            exp = 2 * SafeCast.toUint(value > (1 << 2) - 1);
            value >>= exp;
            result += exp;
            result += SafeCast.toUint(value > 1);
        }
        return result;
    }
    /**
     * @dev Return the log in base 2, following the selected rounding direction, of a positive value.
     * Returns 0 if given 0.
     */
    function log2(uint256 value, Rounding rounding) internal pure returns (uint256) {
        unchecked {
            uint256 result = log2(value);
            return result + SafeCast.toUint(unsignedRoundsUp(rounding) && 1 << result < value);
        }
    }
    /**
     * @dev Return the log in base 10 of a positive value rounded towards zero.
     * Returns 0 if given 0.
     */
    function log10(uint256 value) internal pure returns (uint256) {
        uint256 result = 0;
        unchecked {
            if (value >= 10 ** 64) {
                value /= 10 ** 64;
                result += 64;
            }
            if (value >= 10 ** 32) {
                value /= 10 ** 32;
                result += 32;
```

```
            }
            if (value >= 10 ** 16) {
                value /= 10 ** 16;
                result += 16;
            }
            if (value >= 10 ** 8) {
                value /= 10 ** 8;
                result += 8;
            }
            if (value >= 10 ** 4) {
                value /= 10 ** 4;
                result += 4;
            }
            if (value >= 10 ** 2) {
                value /= 10 ** 2;
                result += 2;
            }
            if (value >= 10 ** 1) {
                result += 1;
            }
        }
        return result;
    }
    /**
     * @dev Return the log in base 10, following the selected rounding direction, of a positive value.
     * Returns 0 if given 0.
     */
    function log10(uint256 value, Rounding rounding) internal pure returns (uint256) {
        unchecked {
            uint256 result = log10(value);
            return result + SafeCast.toUint(unsignedRoundsUp(rounding) && 10 ** result < value);
        }
    }
    /**
     * @dev Return the log in base 256 of a positive value rounded towards zero.
     * Returns 0 if given 0.
     *
     * Adding one to the result gives the number of pairs of hex symbols needed to represent `value` as a hex string.
     */
    function log256(uint256 value) internal pure returns (uint256) {
        uint256 result = 0;
        uint256 isGt;
```

```
        unchecked {
            isGt = SafeCast.toUint(value > (1 << 128) - 1);
            value >>= isGt * 128;
            result += isGt * 16;
            isGt = SafeCast.toUint(value > (1 << 64) - 1);
            value >>= isGt * 64;
            result += isGt * 8;
            isGt = SafeCast.toUint(value > (1 << 32) - 1);
            value >>= isGt * 32;
            result += isGt * 4;
            isGt = SafeCast.toUint(value > (1 << 16) - 1);
            value >>= isGt * 16;
            result += isGt * 2;
            result += SafeCast.toUint(value > (1 << 8) - 1);
        }
        return result;
    }
    /**
     * @dev Return the log in base 256, following the selected rounding direction, of a positive
value.
     * Returns 0 if given 0.
     */
    function log256(uint256 value, Rounding rounding) internal pure returns (uint256) {
        unchecked {
            uint256 result = log256(value);
            return result + SafeCast.toUint(unsignedRoundsUp(rounding) && 1 << (result << 3) <
value);
        }
    }
    /**
     * @dev Returns whether a provided rounding mode is considered rounding up for unsigned
integers.
     */
    function unsignedRoundsUp(Rounding rounding) internal pure returns (bool) {
        return uint8(rounding) % 2 == 1;
    }
}


// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts (last updated v5.1.0) (utils/cryptography/MessageHashUtils.sol)
pragma solidity ^0.8.20;
import {Strings} from "../Strings.sol";
/**
```

```
 * @dev Signature message hash utilities for producing digests to be consumed by {ECDSA}
recovery or signing.
 *
 * The library provides methods for generating a hash of a message that conforms to the
 * https://eips.ethereum.org/EIPS/eip-191[ERC-191] and
https://eips.ethereum.org/EIPS/eip-712[EIP 712]
 * specifications.
 */
library MessageHashUtils {
    /**
     * @dev Returns the keccak256 digest of an ERC-191 signed data with version
     * `0x45` (`personal_sign` messages).
     *
     * The digest is calculated by prefixing a bytes32 `messageHash` with
     * `"\x19Ethereum Signed Message:\n32"` and hashing the result. It corresponds with the
     * hash signed when using the https://eth.wiki/json-rpc/API#eth_sign[`eth_sign`] JSON-RPC
method.
     *
     * NOTE: The `messageHash` parameter is intended to be the result of hashing a raw
message with
     * keccak256, although any bytes32 value can be safely used because the final digest will
     * be re-hashed.
     *
     * See {ECDSA-recover}.
     */
    function toEthSignedMessageHash(bytes32 messageHash) internal pure returns (bytes32
digest) {
        assembly ("memory-safe") {
            mstore(0x00, "\x19Ethereum Signed Message:\n32") // 32 is the bytes-length of
messageHash
            mstore(0x1c, messageHash) // 0x1c (28) is the length of the prefix
            digest := keccak256(0x00, 0x3c) // 0x3c is the length of the prefix (0x1c) +
messageHash (0x20)
        }
    }
    /**
     * @dev Returns the keccak256 digest of an ERC-191 signed data with version
     * `0x45` (`personal_sign` messages).
     *
     * The digest is calculated by prefixing an arbitrary `message` with
     * `"\x19Ethereum Signed Message:\n" + len(message)` and hashing the result. It
corresponds with the
     * hash signed when using the https://eth.wiki/json-rpc/API#eth_sign[`eth_sign`] JSON-RPC
method.
```

```
     *
     * See {ECDSA-recover}.
     */
    function toEthSignedMessageHash(bytes memory message) internal pure returns (bytes32) {
        return
            keccak256(bytes.concat("\x19Ethereum Signed Message:\n",
bytes(Strings.toString(message.length)), message));
    }
    /**
     * @dev Returns the keccak256 digest of an ERC-191 signed data with version
     * `0x00` (data with intended validator).
     *
     * The digest is calculated by prefixing an arbitrary `data` with `"\x19\x00"` and the intended
     * `validator` address. Then hashing the result.
     *
     * See {ECDSA-recover}.
     */
    function toDataWithIntendedValidatorHash(address validator, bytes memory data) internal
pure returns (bytes32) {
        return keccak256(abi.encodePacked(hex"19_00", validator, data));
    }
    /**
     * @dev Returns the keccak256 digest of an EIP-712 typed data (ERC-191 version `0x01`).
     *
     * The digest is calculated from a `domainSeparator` and a `structHash`, by prefixing them
with
     * `\x19\x01` and hashing the result. It corresponds to the hash signed by the
     * https://eips.ethereum.org/EIPS/eip-712[`eth_signTypedData`] JSON-RPC method as part
of EIP-712.
     *
     * See {ECDSA-recover}.
     */
    function toTypedDataHash(bytes32 domainSeparator, bytes32 structHash) internal pure
returns (bytes32 digest) {
        assembly ("memory-safe") {
            let ptr := mload(0x40)
            mstore(ptr, hex"19_01")
            mstore(add(ptr, 0x02), domainSeparator)
            mstore(add(ptr, 0x22), structHash)
            digest := keccak256(ptr, 0x42)
        }
    }
}
```

```
// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts (last updated v5.1.0) (utils/ShortStrings.sol)
pragma solidity ^0.8.20;
import {StorageSlot} from "./StorageSlot.sol";
// | string  |
0xAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
|
// | length  | 0x                                                BB |
type ShortString is bytes32;
/**
 * @dev This library provides functions to convert short memory strings
 * into a `ShortString` type that can be used as an immutable variable.
 *
 * Strings of arbitrary length can be optimized using this library if
 * they are short enough (up to 31 bytes) by packing them with their
 * length (1 byte) in a single EVM word (32 bytes). Additionally, a
 * fallback mechanism can be used for every other case.
 *
 * Usage example:
 *
 * ```solidity
 * contract Named {
 *     using ShortStrings for *;
 *
 *     ShortString private immutable _name;
 *     string private _nameFallback;
 *
 *     constructor(string memory contractName) {
 *         _name = contractName.toShortStringWithFallback(_nameFallback);
 *     }
 *
 *     function name() external view returns (string memory) {
 *         return _name.toStringWithFallback(_nameFallback);
 *     }
 * }
 * ```
 */
library ShortStrings {
    // Used as an identifier for strings longer than 31 bytes.
    bytes32 private constant FALLBACK_SENTINEL =
0x00000000000000000000000000000000000000000000000000000000000000FF;
    error StringTooLong(string str);
    error InvalidShortString();
```

```
    /**
     * @dev Encode a string of at most 31 chars into a `ShortString`.
     *
     * This will trigger a `StringTooLong` error is the input string is too long.
     */
    function toShortString(string memory str) internal pure returns (ShortString) {
        bytes memory bstr = bytes(str);
        if (bstr.length > 31) {
            revert StringTooLong(str);
        }
        return ShortString.wrap(bytes32(uint256(bytes32(bstr)) | bstr.length));
    }
    /**
     * @dev Decode a `ShortString` back to a "normal" string.
     */
    function toString(ShortString sstr) internal pure returns (string memory) {
        uint256 len = byteLength(sstr);
        // using `new string(len)` would work locally but is not memory safe.
        string memory str = new string(32);
        assembly ("memory-safe") {
            mstore(str, len)
            mstore(add(str, 0x20), sstr)
        }
        return str;
    }
    /**
     * @dev Return the length of a `ShortString`.
     */
    function byteLength(ShortString sstr) internal pure returns (uint256) {
        uint256 result = uint256(ShortString.unwrap(sstr)) & 0xFF;
        if (result > 31) {
            revert InvalidShortString();
        }
        return result;
    }
    /**
     * @dev Encode a string into a `ShortString`, or write it to storage if it is too long.
     */
    function toShortStringWithFallback(string memory value, string storage store) internal returns
(ShortString) {
        if (bytes(value).length < 32) {
            return toShortString(value);
        } else {
            StorageSlot.getStringSlot(store).value = value;
```

```
            return ShortString.wrap(FALLBACK_SENTINEL);
        }
    }
    /**
     * @dev Decode a string that was encoded to `ShortString` or written to storage using
{setWithFallback}.
     */
    function toStringWithFallback(ShortString value, string storage store) internal pure returns
(string memory) {
        if (ShortString.unwrap(value) != FALLBACK_SENTINEL) {
            return toString(value);
        } else {
            return store;
        }
    }
    /**
     * @dev Return the length of a string that was encoded to `ShortString` or written to storage
using
     * {setWithFallback}.
     *
     * WARNING: This will return the "byte length" of the string. This may not reflect the actual
length in terms of
     * actual characters as the UTF-8 encoding of a single character can span over multiple
bytes.
     */
    function byteLengthWithFallback(ShortString value, string storage store) internal view returns
(uint256) {
        if (ShortString.unwrap(value) != FALLBACK_SENTINEL) {
            return byteLength(value);
        } else {
            return bytes(store).length;
        }
    }
}


// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts (last updated v5.0.0) (interfaces/IERC5267.sol)
pragma solidity ^0.8.20;
interface IERC5267 {
    /**
     * @dev MAY be emitted to signal that the domain could have changed.
     */
    event EIP712DomainChanged();
```

```
    /**
     * @dev returns the fields and values that describe the domain separator used by this
contract for EIP-712
     * signature.
     */
    function eip712Domain()
        external
        view
        returns (
            bytes1 fields,
            string memory name,
            string memory version,
            uint256 chainId,
            address verifyingContract,
            bytes32 salt,
            uint256[] memory extensions
        );
}


// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts (last updated v5.0.0) (governance/utils/IVotes.sol)
pragma solidity ^0.8.20;
/**
 * @dev Common interface for {ERC20Votes}, {ERC721Votes}, and other {Votes}-enabled
contracts.
 */
interface IVotes {
    /**
     * @dev The signature used has expired.
     */
    error VotesExpiredSignature(uint256 expiry);
    /**
     * @dev Emitted when an account changes their delegate.
     */
    event DelegateChanged(address indexed delegator, address indexed fromDelegate, address
indexed toDelegate);
    /**
     * @dev Emitted when a token transfer or delegate change results in changes to a delegate's
number of voting units.
     */
    event DelegateVotesChanged(address indexed delegate, uint256 previousVotes, uint256
newVotes);
    /**
```

```
     * @dev Returns the current amount of votes that `account` has.
     */
    function getVotes(address account) external view returns (uint256);
    /**
     * @dev Returns the amount of votes that `account` had at a specific moment in the past. If the `clock()` is
     * configured to use block numbers, this will return the value at the end of the corresponding block.
     */
    function getPastVotes(address account, uint256 timepoint) external view returns (uint256);
    /**
     * @dev Returns the total supply of votes available at a specific moment in the past. If the `clock()` is
     * configured to use block numbers, this will return the value at the end of the corresponding block.
     *
     * NOTE: This value is the sum of all available votes, which is not necessarily the sum of all delegated votes.
     * Votes that have not been delegated are still part of total supply, even though they would not participate in a
     * vote.
     */
    function getPastTotalSupply(uint256 timepoint) external view returns (uint256);
    /**
     * @dev Returns the delegate that `account` has chosen.
     */
    function delegates(address account) external view returns (address);
    /**
     * @dev Delegates votes from the sender to `delegatee`.
     */
    function delegate(address delegatee) external;
    /**
     * @dev Delegates votes from signer to `delegatee`.
     */
    function delegateBySig(address delegatee, uint256 nonce, uint256 expiry, uint8 v, bytes32 r, bytes32 s) external;
}


// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts (last updated v5.0.0) (interfaces/IERC6372.sol)
pragma solidity ^0.8.20;
interface IERC6372 {
    /**
```

```
     * @dev Clock used for flagging checkpoints. Can be overridden to implement timestamp
based checkpoints (and voting).
     */
    function clock() external view returns (uint48);
    /**
     * @dev Description of the clock
     */
    // solhint-disable-next-line func-name-mixedcase
    function CLOCK_MODE() external view returns (string memory);
}


// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts (last updated v5.1.0) (utils/Panic.sol)
pragma solidity ^0.8.20;
/**
 * @dev Helper library for emitting standardized panic codes.
 *
 * ```solidity
 * contract Example {
 *      using Panic for uint256;
 *
 *      // Use any of the declared internal constants
 *      function foo() { Panic.GENERIC.panic(); }
 *
 *      // Alternatively
 *      function foo() { Panic.panic(Panic.GENERIC); }
 * }
 * ```
 *
 * Follows the list from
https://github.com/ethereum/solidity/blob/v0.8.24/libsolutil/ErrorCodes.h[libsolutil].
 *
 * _Available since v5.1._
 */
// slither-disable-next-line unused-state
library Panic {
    /// @dev generic / unspecified error
    uint256 internal constant GENERIC = 0x00;
    /// @dev used by the assert() builtin
    uint256 internal constant ASSERT = 0x01;
    /// @dev arithmetic underflow or overflow
    uint256 internal constant UNDER_OVERFLOW = 0x11;
    /// @dev division or modulo by zero
```

```
    uint256 internal constant DIVISION_BY_ZERO = 0x12;
    /// @dev enum conversion error
    uint256 internal constant ENUM_CONVERSION_ERROR = 0x21;
    /// @dev invalid encoding in storage
    uint256 internal constant STORAGE_ENCODING_ERROR = 0x22;
    /// @dev empty array pop
    uint256 internal constant EMPTY_ARRAY_POP = 0x31;
    /// @dev array out of bounds access
    uint256 internal constant ARRAY_OUT_OF_BOUNDS = 0x32;
    /// @dev resource error (too large allocation or too large array)
    uint256 internal constant RESOURCE_ERROR = 0x41;
    /// @dev calling invalid internal function
    uint256 internal constant INVALID_INTERNAL_FUNCTION = 0x51;
    /// @dev Reverts with a panic code. Recommended to use with
    /// the internal constants with predefined codes.
    function panic(uint256 code) internal pure {
        assembly ("memory-safe") {
            mstore(0x00, 0x4e487b71)
            mstore(0x20, code)
            revert(0x1c, 0x24)
        }
    }
}


// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts (last updated v5.1.0) (utils/Strings.sol)
pragma solidity ^0.8.20;
import {Math} from "./math/Math.sol";
import {SignedMath} from "./math/SignedMath.sol";
/**
 * @dev String operations.
 */
library Strings {
    bytes16 private constant HEX_DIGITS = "0123456789abcdef";
    uint8 private constant ADDRESS_LENGTH = 20;
    /**
     * @dev The `value` string doesn't fit in the specified `length`.
     */
    error StringsInsufficientHexLength(uint256 value, uint256 length);
    /**
     * @dev Converts a `uint256` to its ASCII `string` decimal representation.
     */
    function toString(uint256 value) internal pure returns (string memory) {
```

```
        unchecked {
            uint256 length = Math.log10(value) + 1;
            string memory buffer = new string(length);
            uint256 ptr;
            assembly ("memory-safe") {
                ptr := add(buffer, add(32, length))
            }
            while (true) {
                ptr--;
                assembly ("memory-safe") {
                    mstore8(ptr, byte(mod(value, 10), HEX_DIGITS))
                }
                value /= 10;
                if (value == 0) break;
            }
            return buffer;
        }
    }
    /**
     * @dev Converts a `int256` to its ASCII `string` decimal representation.
     */
    function toStringSigned(int256 value) internal pure returns (string memory) {
        return string.concat(value < 0 ? "-" : "", toString(SignedMath.abs(value)));
    }
    /**
     * @dev Converts a `uint256` to its ASCII `string` hexadecimal representation.
     */
    function toHexString(uint256 value) internal pure returns (string memory) {
        unchecked {
            return toHexString(value, Math.log256(value) + 1);
        }
    }
    /**
     * @dev Converts a `uint256` to its ASCII `string` hexadecimal representation with fixed length.
     */
    function toHexString(uint256 value, uint256 length) internal pure returns (string memory) {
        uint256 localValue = value;
        bytes memory buffer = new bytes(2 * length + 2);
        buffer[0] = "0";
        buffer[1] = "x";
        for (uint256 i = 2 * length + 1; i > 1; --i) {
            buffer[i] = HEX_DIGITS[localValue & 0xf];
            localValue >>= 4;
```

```
        }
        if (localValue != 0) {
            revert StringsInsufficientHexLength(value, length);
        }
        return string(buffer);
    }
    /**
     * @dev Converts an `address` with fixed length of 20 bytes to its not checksummed ASCII
`string` hexadecimal
     * representation.
     */
    function toHexString(address addr) internal pure returns (string memory) {
        return toHexString(uint256(uint160(addr)), ADDRESS_LENGTH);
    }
    /**
     * @dev Converts an `address` with fixed length of 20 bytes to its checksummed ASCII
`string` hexadecimal
     * representation, according to EIP-55.
     */
    function toChecksumHexString(address addr) internal pure returns (string memory) {
        bytes memory buffer = bytes(toHexString(addr));
        // hash the hex part of buffer (skip length + 2 bytes, length 40)
        uint256 hashValue;
        assembly ("memory-safe") {
            hashValue := shr(96, keccak256(add(buffer, 0x22), 40))
        }
        for (uint256 i = 41; i > 1; --i) {
            // possible values for buffer[i] are 48 (0) to 57 (9) and 97 (a) to 102 (f)
            if (hashValue & 0xf > 7 && uint8(buffer[i]) > 96) {
                // case shift by xoring with 0x20
                buffer[i] ^= 0x20;
            }
            hashValue >>= 4;
        }
        return string(buffer);
    }
    /**
     * @dev Returns true if the two strings are equal.
     */
    function equal(string memory a, string memory b) internal pure returns (bool) {
        return bytes(a).length == bytes(b).length && keccak256(bytes(a)) == keccak256(bytes(b));
    }
}
```

```
// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts (last updated v5.1.0) (utils/StorageSlot.sol)
// This file was procedurally generated from scripts/generate/templates/StorageSlot.js.
pragma solidity ^0.8.20;
/**
 * @dev Library for reading and writing primitive types to specific storage slots.
 *
 * Storage slots are often used to avoid storage conflict when dealing with upgradeable contracts.
 * This library helps with reading and writing to such slots without the need for inline assembly.
 *
 * The functions in this library return Slot structs that contain a `value` member that can be used to read or write.
 *
 * Example usage to set ERC-1967 implementation slot:
 * ```solidity
 * contract ERC1967 {
 *     // Define the slot. Alternatively, use the SlotDerivation library to derive the slot.
 *     bytes32 internal constant _IMPLEMENTATION_SLOT = 0x360894a13ba1a3210667c828492db98dca3e2076cc3735a920a3ca505d382bbc;
 *
 *     function _getImplementation() internal view returns (address) {
 *         return StorageSlot.getAddressSlot(_IMPLEMENTATION_SLOT).value;
 *     }
 *
 *     function _setImplementation(address newImplementation) internal {
 *         require(newImplementation.code.length > 0);
 *         StorageSlot.getAddressSlot(_IMPLEMENTATION_SLOT).value = newImplementation;
 *     }
 * }
 * ```
 *
 * TIP: Consider using this library along with {SlotDerivation}.
 */
library StorageSlot {
    struct AddressSlot {
        address value;
    }
    struct BooleanSlot {
        bool value;
    }
    struct Bytes32Slot {
        bytes32 value;
```

```
}
struct Uint256Slot {
    uint256 value;
}
struct Int256Slot {
    int256 value;
}
struct StringSlot {
    string value;
}
struct BytesSlot {
    bytes value;
}
/**
 * @dev Returns an `AddressSlot` with member `value` located at `slot`.
 */
function getAddressSlot(bytes32 slot) internal pure returns (AddressSlot storage r) {
    assembly ("memory-safe") {
        r.slot := slot
    }
}
/**
 * @dev Returns a `BooleanSlot` with member `value` located at `slot`.
 */
function getBooleanSlot(bytes32 slot) internal pure returns (BooleanSlot storage r) {
    assembly ("memory-safe") {
        r.slot := slot
    }
}
/**
 * @dev Returns a `Bytes32Slot` with member `value` located at `slot`.
 */
function getBytes32Slot(bytes32 slot) internal pure returns (Bytes32Slot storage r) {
    assembly ("memory-safe") {
        r.slot := slot
    }
}
/**
 * @dev Returns a `Uint256Slot` with member `value` located at `slot`.
 */
function getUint256Slot(bytes32 slot) internal pure returns (Uint256Slot storage r) {
    assembly ("memory-safe") {
        r.slot := slot
    }
```

```
    }
    /**
     * @dev Returns a `Int256Slot` with member `value` located at `slot`.
     */
    function getInt256Slot(bytes32 slot) internal pure returns (Int256Slot storage r) {
        assembly ("memory-safe") {
            r.slot := slot
        }
    }
    /**
     * @dev Returns a `StringSlot` with member `value` located at `slot`.
     */
    function getStringSlot(bytes32 slot) internal pure returns (StringSlot storage r) {
        assembly ("memory-safe") {
            r.slot := slot
        }
    }
    /**
     * @dev Returns an `StringSlot` representation of the string storage pointer `store`.
     */
    function getStringSlot(string storage store) internal pure returns (StringSlot storage r) {
        assembly ("memory-safe") {
            r.slot := store.slot
        }
    }
    /**
     * @dev Returns a `BytesSlot` with member `value` located at `slot`.
     */
    function getBytesSlot(bytes32 slot) internal pure returns (BytesSlot storage r) {
        assembly ("memory-safe") {
            r.slot := slot
        }
    }
    /**
     * @dev Returns an `BytesSlot` representation of the bytes storage pointer `store`.
     */
    function getBytesSlot(bytes storage store) internal pure returns (BytesSlot storage r) {
        assembly ("memory-safe") {
            r.slot := store.slot
        }
    }
}
```

```
// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts (last updated v5.1.0) (utils/math/SignedMath.sol)
pragma solidity ^0.8.20;
import {SafeCast} from "./SafeCast.sol";
/**
 * @dev Standard signed math utilities missing in the Solidity language.
 */
library SignedMath {
    /**
     * @dev Branchless ternary evaluation for `a ? b : c`. Gas costs are constant.
     *
     * IMPORTANT: This function may reduce bytecode size and consume less gas when used standalone.
     * However, the compiler may optimize Solidity ternary operations (i.e. `a ? b : c`) to only compute
     * one branch when needed, making this function more expensive.
     */
    function ternary(bool condition, int256 a, int256 b) internal pure returns (int256) {
        unchecked {
            // branchless ternary works because:
            // b ^ (a ^ b) == a
            // b ^ 0 == b
            return b ^ ((a ^ b) * int256(SafeCast.toUint(condition)));
        }
    }
    /**
     * @dev Returns the largest of two signed numbers.
     */
    function max(int256 a, int256 b) internal pure returns (int256) {
        return ternary(a > b, a, b);
    }
    /**
     * @dev Returns the smallest of two signed numbers.
     */
    function min(int256 a, int256 b) internal pure returns (int256) {
        return ternary(a < b, a, b);
    }
    /**
     * @dev Returns the average of two signed numbers without overflow.
     * The result is rounded towards zero.
     */
    function average(int256 a, int256 b) internal pure returns (int256) {
        // Formula from the book "Hacker's Delight"
        int256 x = (a & b) + ((a ^ b) >> 1);
```

```
        return x + (int256(uint256(x) >> 255) & (a ^ b));
    }
    /**
     * @dev Returns the absolute unsigned value of a signed value.
     */
    function abs(int256 n) internal pure returns (uint256) {
        unchecked {
            // Formula from the "Bit Twiddling Hacks" by Sean Eron Anderson.
            // Since `n` is a signed integer, the generated bytecode will use the SAR opcode to perform the right shift,
            // taking advantage of the most significant (or "sign" bit) in two's complement representation.
            // This opcode adds new most significant bits set to the value of the previous most significant bit. As a result,
            // the mask will either be `bytes32(0)` (if n is positive) or `~bytes32(0)` (if n is negative).
            int256 mask = n >> 255;
            // A `bytes32(0)` mask leaves the input unchanged, while a `~bytes32(0)` mask complements it.
            return uint256((n + mask) ^ mask);
        }
    }
}
```