```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.20;


// OpenZeppelin v5 imports

import {ERC20} from "@openzeppelin/contracts/token/ERC20/ERC20.sol";

import {ERC20Permit} from
"@openzeppelin/contracts/token/ERC20/extensions/ERC20Permit.sol";

import {ERC20Votes} from
"@openzeppelin/contracts/token/ERC20/extensions/ERC20Votes.sol";

import {ERC20Burnable} from
"@openzeppelin/contracts/token/ERC20/extensions/ERC20Burnable.sol";

import {AccessControl} from "@openzeppelin/contracts/access/AccessControl.sol";

import {Pausable} from "@openzeppelin/contracts/utils/Pausable.sol";

import {SafeERC20, IERC20} from
"@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";


/**

 * @title AXM – Axiom Token (peaq-ready)

 * @notice Fixed-supply governance token with optional DID/KYC gating and

 *         compliance controls designed for peaq’s EVM environment.

 */

contract AXM is

   ERC20,

   ERC20Permit,

   ERC20Votes,

   ERC20Burnable,

   AccessControl,

   Pausable

{

   using SafeERC20 for IERC20;


   // ===== Roles =====

   bytes32 public constant PAUSER_ROLE     = keccak256("PAUSER_ROLE");

   bytes32 public constant COMPLIANCE_ROLE = keccak256("COMPLIANCE_ROLE");
```

```
bytes32 public constant RESCUER_ROLE    = keccak256("RESCUER_ROLE");

// ===== Supply (15,000,000,000 * 1e18) =====
uint256 public constant TOTAL_SUPPLY = 15_000_000_000 ether;

// ===== Compliance / Registry =====
/// @dev Minimal interface to an on-chain identity/KYC registry.
/// Hook this to a peaq DID/KYC contract or your own registry.
interface IIdentityRegistry {
    function isVerified(address user) external view returns (bool);
}

IIdentityRegistry public identityRegistry;

/// @dev Global switch: if true, both sender & receiver must be verified.
bool public kycEnforced;

/// @dev Per-address blocklist (hard stop).
mapping(address => bool) public blocked;

/// @dev Optional allowlist mode: when enabled, only allowlisted can transfer/receive.
bool public allowlistEnforced;
mapping(address => bool) public allowlisted;

// ===== Events =====
event InitialSupplyMinted(address indexed vault, uint256 amount);
event IdentityRegistrySet(address indexed registry);
event KYCEnforcementSet(bool enforced);
event AllowlistEnforcementSet(bool enforced);
event Blocked(address indexed account, bool isBlocked);
event Allowlisted(address indexed account, bool isAllowlisted);

constructor(address distributionVault, address admin)
```

```
    ERC20("Axiom", "AXM")
    ERC20Permit("Axiom")
{
    require(distributionVault != address(0), "AXM: vault is zero");
    require(admin != address(0), "AXM: admin is zero");

    // Roles
    _grantRole(DEFAULT_ADMIN_ROLE, admin);
    _grantRole(PAUSER_ROLE, admin);
    _grantRole(COMPLIANCE_ROLE, admin);
    _grantRole(RESCUER_ROLE, admin);

    // One-time mint to vault
    _mint(distributionVault, TOTAL_SUPPLY);
    emit InitialSupplyMinted(distributionVault, TOTAL_SUPPLY);
}

// ===== Admin & Compliance controls =====

function pause() external onlyRole(PAUSER_ROLE) { _pause(); }
function unpause() external onlyRole(PAUSER_ROLE) { _unpause(); }

function setIdentityRegistry(address registry) external onlyRole(COMPLIANCE_ROLE) {
    identityRegistry = IIdentityRegistry(registry);
    emit IdentityRegistrySet(registry);
}

function setKYCEnforcement(bool enforced) external onlyRole(COMPLIANCE_ROLE) {
    kycEnforced = enforced;
    emit KYCEnforcementSet(enforced);
}

function setAllowlistEnforcement(bool enforced) external onlyRole(COMPLIANCE_ROLE) {
        allowlistEnforced = enforced;

        emit AllowlistEnforcementSet(enforced);

    }


    function setBlocked(address account, bool isBlocked) external
onlyRole(COMPLIANCE_ROLE) {

        blocked[account] = isBlocked;

        emit Blocked(account, isBlocked);

    }


    function setAllowlisted(address account, bool isAllow) external
onlyRole(COMPLIANCE_ROLE) {

        allowlisted[account] = isAllow;

        emit Allowlisted(account, isAllow);

    }


    /**

     * @notice Rescue unrelated ERC20s accidentally sent to this contract.

     *         Cannot pull AXM itself. Only RESCUER_ROLE can call.

     */

    function rescueTokens(address token, address to, uint256 amount)

        external

        onlyRole(RESCUER_ROLE)

    {

        require(token != address(this), "AXM: cannot rescue AXM");

        require(to != address(0), "AXM: to is zero");

        IERC20(token).safeTransfer(to, amount);

    }


    // ===== Transfer gate (OZ v5 uses _update) =====


    function _update(address from, address to, uint256 value)

        internal
        override(ERC20, ERC20Votes)
        whenNotPaused
    {
        // Minting to != 0 and burning from != 0 will still pass through these gates.
        if (from != address(0)) {
            require(!blocked[from], "AXM: sender blocked");
        }
        if (to != address(0)) {
            require(!blocked[to], "AXM: recipient blocked");
        }

        if (allowlistEnforced) {
            // When enabled, both sides must be on allowlist (except mint/burn)
            if (from != address(0)) { require(allowlisted[from], "AXM: sender not allowlisted"); }
            if (to   != address(0)) { require(allowlisted[to],   "AXM: recipient not allowlisted"); }
        }

        if (kycEnforced) {
            // Identity registry must be set
            require(address(identityRegistry) != address(0), "AXM: registry not set");
            if (from != address(0)) { require(identityRegistry.isVerified(from), "AXM: sender
unverified"); }
            if (to   != address(0)) { require(identityRegistry.isVerified(to),   "AXM: recipient
unverified"); }
        }

        super._update(from, to, value);
    }

    // ===== Required overrides for ERC20Votes (compiles with OZ v5) =====

    function _afterTokenTransfer(address from, address to, uint256 amount)
        internal
        override(ERC20, ERC20Votes)
    { super._afterTokenTransfer(from, to, amount); }


    function _mint(address to, uint256 amount)
        internal
        override(ERC20, ERC20Votes)
    { super._mint(to, amount); }


    function _burn(address from, uint256 amount)
        internal
        override(ERC20, ERC20Votes)
    { super._burn(from, amount); }
}
```