



Monetec-Token-Vertrag (MTC)

0x605380d0CD32746FBBe31f3F9A5a367EB92Bd5C0

Privatplatzierung

€100.000,00 Mindestinvestition; für professionelle Investoren

Monetec GmbH, Mai 2024


```

1  /**
2   *Submitted for verification at Etherscan.io on 2024-05-28
3   */
4
5   // SPDX-License-Identifier: MIT
6
7   pragma solidity 0.8.24;
8
9   abstract contract DomainAware {
10
11       // Mapping of ChainID to domain separators. This is a very gas efficient way
12       // to not recalculate the domain separator on every call, while still
13       // automatically detecting ChainID changes.
14       mapping(uint256 => bytes32) private domainSeparators;
15
16       constructor() {
17         _updateDomainSeparator();
18       }
19
20       function domainName() public virtual view returns (string memory);
21
22       function domainVersion() public virtual view returns (string memory);
23
24       function generateDomainSeparator() public view returns (bytes32) {
25         uint256 chainID = _chainID();
26
27         // no need for assembly, running very rarely
28         bytes32 domainSeparatorHash = keccak256(
29           abi.encode(
30             keccak256(
31               "EIP712Domain(string name,string version,uint256 chainId,address verifyingContract)"
32             ),
33             keccak256(bytes(domainName())), // ERC-20 Name
34             keccak256(bytes(domainVersion())), // Version
35             chainID,
36             address(this)
37           )
38         );
39
40         return domainSeparatorHash;
41       }
42
43       function domainSeparator() external returns (bytes32) {
44         return _domainSeparator();
45       }
46
47       function _updateDomainSeparator() private returns (bytes32) {
48         uint256 chainID = _chainID();
49
50         bytes32 newDomainSeparator = generateDomainSeparator();

```

```

51
52     domainSeparators[chainID] = newDomainSeparator;
53
54     return newDomainSeparator;
55 }
56
57 // Returns the domain separator, updating it if chainID changes
58 function _domainSeparator() private returns (bytes32) {
59     bytes32 currentDomainSeparator = domainSeparators[_chainID()];
60
61     if (currentDomainSeparator != 0x00) {
62         return currentDomainSeparator;
63     }
64
65     return _updateDomainSeparator();
66 }
67
68 function _chainID() internal view returns (uint256) {
69     uint256 chainID;
70     assembly {
71         chainID := chainid()
72     }
73
74     return chainID;
75 }
76 }
77
78 /**
79  * @title IERC1400TokensRecipient
80  * @dev ERC1400TokensRecipient interface
81  */
82 interface IERC1400TokensRecipient {
83
84     function canReceive(
85         bytes calldata payload,
86         bytes32 partition,
87         address operator,
88         address from,
89         address to,
90         uint value,
91         bytes calldata data,
92         bytes calldata operatorData
93     ) external view returns(bool);
94
95     function tokensReceived(
96         bytes calldata payload,
97         bytes32 partition,
98         address operator,
99         address from,
100         address to,

```

```

101         uint value,
102         bytes calldata data,
103         bytes calldata operatorData
104     ) external;
105
106 }
107
108 /**
109  * @title IERC1400TokensSender
110  * @dev ERC1400TokensSender interface
111  */
112 interface IERC1400TokensSender {
113
114     function canTransfer(
115         bytes calldata payload,
116         bytes32 partition,
117         address operator,
118         address from,
119         address to,
120         uint value,
121         bytes calldata data,
122         bytes calldata operatorData
123     ) external view returns(bool);
124
125     function tokensToTransfer(
126         bytes calldata payload,
127         bytes32 partition,
128         address operator,
129         address from,
130         address to,
131         uint value,
132         bytes calldata data,
133         bytes calldata operatorData
134     ) external;
135
136 }
137
138 /**
139  * @title IERC1400TokensChecker
140  * @dev IERC1400TokensChecker interface
141  */
142 interface IERC1400TokensChecker {
143
144     // function canTransfer(
145     //     bytes calldata payload,
146     //     address operator,
147     //     address from,
148     //     address to,
149     //     uint256 value,
150     //     bytes calldata data,

```



```

151 // bytes calldata operatorData
152 // ) external view returns (byte, bytes32);
153
154 function canTransferByPartition(
155     bytes calldata payload,
156     bytes32 partition,
157     address operator,
158     address from,
159     address to,
160     uint256 value,
161     bytes calldata data,
162     bytes calldata operatorData
163 ) external view returns (bytes1, bytes32, bytes32);
164
165 }
166
167 /**
168  * @title IERC1400TokensValidator
169  * @dev ERC1400TokensValidator interface
170  */
171 interface IERC1400TokensValidator {
172
173     /**
174      * @dev Verify if a token transfer can be executed or not, on the validator's perspective.
175      * @param token Token address.
176      * @param payload Payload of the initial transaction.
177      * @param partition Name of the partition (left empty for ERC20 transfer).
178      * @param operator Address which triggered the balance decrease (through transfer or redemption).
179      * @param from Token holder.
180      * @param to Token recipient for a transfer and 0x for a redemption.
181      * @param value Number of tokens the token holder balance is decreased by.
182      * @param data Extra information.
183      * @param operatorData Extra information, attached by the operator (if any).
184      * @return 'true' if the token transfer can be validated, 'false' if not.
185      */
186     struct ValidateData {
187         address token;
188         bytes payload;
189         bytes32 partition;
190         address operator;
191         address from;
192         address to;
193         uint value;
194         bytes data;
195         bytes operatorData;
196     }
197
198     function canValidate(ValidateData calldata data) external view returns(bool);
199
200     function tokensToValidate(

```



```

201     bytes calldata payload,
202     bytes32 partition,
203     address operator,
204     address from,
205     address to,
206     uint value,
207     bytes calldata data,
208     bytes calldata operatorData
209 ) external;
210
211 }
212
213 /**
214  * @title Roles
215  * @dev Library for managing addresses assigned to a Role.
216  */
217 library Roles {
218     struct Role {
219         mapping (address => bool) bearer;
220     }
221
222     /**
223      * @dev Give an account access to this role.
224      */
225     function add(Role storage role, address account) internal {
226         require(!has(role, account), "Roles: account already has role");
227         role.bearer[account] = true;
228     }
229
230     /**
231      * @dev Remove an account's access to this role.
232      */
233     function remove(Role storage role, address account) internal {
234         require(has(role, account), "Roles: account does not have role");
235         role.bearer[account] = false;
236     }
237
238     /**
239      * @dev Check if an account has this role.
240      * @return bool
241      */
242     function has(Role storage role, address account) internal view returns (bool) {
243         require(account != address(0), "Roles: account is the zero address");
244         return role.bearer[account];
245     }
246 }
247
248 /**
249  * @title MinterRole
250  * @dev Minters are responsible for minting new tokens.

```



```

251 */
252 contract MinterRole {
253     using Roles for Roles.Role;
254
255     event MinterAdded(address indexed account);
256     event MinterRemoved(address indexed account);
257
258     Roles.Role private _minters;
259
260     constructor() {
261         _addMinter(msg.sender);
262     }
263
264     modifier onlyMinter() virtual {
265         require(isMinter(msg.sender));
266         _;
267     }
268
269     function isMinter(address account) public view returns (bool) {
270         return _minters.has(account);
271     }
272
273     function addMinter(address account) external onlyMinter {
274         _addMinter(account);
275     }
276
277     function removeMinter(address account) external onlyMinter {
278         _removeMinter(account);
279     }
280
281     function renounceMinter() external {
282         _removeMinter(msg.sender);
283     }
284
285     function _addMinter(address account) internal {
286         _minters.add(account);
287         emit MinterAdded(account);
288     }
289
290     function _removeMinter(address account) internal {
291         _minters.remove(account);
292         emit MinterRemoved(account);
293     }
294 }
295
296 contract ERC1820Implementer {
297     bytes32 constant ERC1820_ACCEPT_MAGIC = keccak256(abi.encodePacked("ERC1820_ACCEPT_MAGIC"));
298
299     mapping(bytes32 => bool) internal _interfaceHashes;
300

```

```

301 function canImplementInterfaceForAddress(bytes32 interfaceHash, address /*addr*/) // Comments to avoid compilation warnings for unused variables.
302 external
303 view
304 returns(bytes32)
305 {
306     if(_interfaceHashes[interfaceHash]) {
307         return ERC1820_ACCEPT_MAGIC;
308     } else {
309         return "";
310     }
311 }
312
313 function _setInterface(string memory interfaceLabel) internal {
314     _interfaceHashes[keccak256(abi.encodePacked(interfaceLabel))] = true;
315 }
316
317 }
318
319 interface IERC1820Registry {
320     event InterfacImplementerSet(address indexed account, bytes32 indexed interfaceHash, address indexed implementer);
321
322     event ManagerChanged(address indexed account, address indexed newManager);
323
324     /**
325      * @dev Sets `newManager` as the manager for `account`. A manager of an
326      * account is able to set interface implementers for it.
327      *
328      * By default, each account is its own manager. Passing a value of `0x0` in
329      * `newManager` will reset the manager to this initial state.
330      *
331      * Emits a {ManagerChanged} event.
332      *
333      * Requirements:
334      *
335      * - the caller must be the current manager for `account`.
336      */
337     function setManager(address account, address newManager) external;
338
339     /**
340      * @dev Returns the manager for `account`.
341      *
342      * See {setManager}.
343      */
344     function getManager(address account) external view returns (address);
345
346     /**
347      * @dev Sets the `implementer` contract as ``account``'s implementer for
348      * `interfaceHash`.
349      *
350      * `account` being the zero address is an alias for the caller's address.

```



```

351 * The zero address can also be used in `implementer` to remove an old one.
352 *
353 * See {interfaceHash} to learn how these are created.
354 *
355 * Emits an {InterfaceImplementerSet} event.
356 *
357 * Requirements:
358 *
359 * - the caller must be the current manager for `account`.
360 * - `interfaceHash` must not be an {IERC165} interface id (i.e. it must not
361 *   end in 28 zeroes).
362 * - `implementer` must implement {IERC1820Implementer} and return true when
363 *   queried for support, unless `implementer` is the caller. See
364 *   {IERC1820Implementer-canImplementInterfaceForAddress}.
365 */
366 function setInterfaceImplementer(
367     address account,
368     bytes32 _interfaceHash,
369     address implementer
370 ) external;
371
372 /**
373  * @dev Returns the implementer of `interfaceHash` for `account`. If no such
374  * implementer is registered, returns the zero address.
375  *
376  * If `interfaceHash` is an {IERC165} interface id (i.e. it ends with 28
377  * zeroes), `account` will be queried for support of it.
378  *
379  * `account` being the zero address is an alias for the caller's address.
380  */
381 function getInterfaceImplementer(address account, bytes32 _interfaceHash) external view returns (address);
382
383 /**
384  * @dev Returns the interface hash for an `interfaceName`, as defined in the
385  * corresponding
386  * https://eips.ethereum.org/EIPS/eip-1820#interface-name[section of the EIP].
387  */
388 function interfaceHash(string calldata interfaceName) external pure returns (bytes32);
389
390 /**
391  * @notice Updates the cache with whether the contract implements an ERC165 interface or not.
392  * @param account Address of the contract for which to update the cache.
393  * @param interfaceId ERC165 interface for which to update the cache.
394  */
395 function updateERC165Cache(address account, bytes4 interfaceId) external;
396
397 /**
398  * @notice Checks whether a contract implements an ERC165 interface or not.
399  * If the result is not cached a direct lookup on the contract address is performed.
400  * If the result is not cached or the cached value is out-of-date, the cache MUST be updated manually by calling

```

```

401 * {updateERC165Cache} with the contract address.
402 * @param account Address of the contract to check.
403 * @param interfaceId ERC165 interface to check.
404 * @return True if `account` implements `interfaceId`, false otherwise.
405 */
406 function implementsERC165Interface(address account, bytes4 interfaceId) external view returns (bool);
407
408 /**
409  * @notice Checks whether a contract implements an ERC165 interface or not without using nor updating the cache.
410  * @param account Address of the contract to check.
411  * @param interfaceId ERC165 interface to check.
412  * @return True if `account` implements `interfaceId`, false otherwise.
413  */
414 function implementsERC165InterfaceNoCache(address account, bytes4 interfaceId) external view returns (bool);
415 }
416
417 contract ERC1820Client {
418     IERC1820Registry constant ERC1820REGISTRY = IERC1820Registry(0x1820a4B7618BdE71Dce8cdc73aAB6C95905faD24);
419
420     function setInterfaceImplementation(string memory _interfaceLabel, address _implementation) internal {
421         bytes32 interfaceHash = keccak256(abi.encodePacked(_interfaceLabel));
422         ERC1820REGISTRY.setInterfaceImplementer(address(this), interfaceHash, _implementation);
423     }
424
425     function interfaceAddr(address addr, string memory _interfaceLabel) internal view returns(address) {
426         bytes32 interfaceHash = keccak256(abi.encodePacked(_interfaceLabel));
427         return ERC1820REGISTRY.getInterfaceImplementer(addr, interfaceHash);
428     }
429
430
431 }
432
433 abstract contract Context {
434     function _msgSender() internal view returns (address) {
435         return msg.sender;
436     }
437 }
438
439 abstract contract Ownable is Context {
440     address private _owner;
441
442     event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);
443
444     /**
445      * @dev Initializes the contract setting the deployer as the initial owner.
446      */
447     constructor() {
448         _transferOwnership(_msgSender());
449     }
450

```



```

451  /**
452  * @dev Throws if called by any account other than the owner.
453  */
454  modifier onlyOwner() {
455      _checkOwner();
456      _;
457  }
458
459  /**
460  * @dev Returns the address of the current owner.
461  */
462  function owner() public view returns (address) {
463      return _owner;
464  }
465
466  /**
467  * @dev Throws if the sender is not the owner.
468  */
469  function _checkOwner() internal view {
470      require(owner() == _msgSender(), "Ownable: caller is not the owner");
471  }
472
473  /**
474  * @dev Leaves the contract without owner. It will not be possible to call
475  * `onlyOwner` functions anymore. Can only be called by the current owner.
476  *
477  * NOTE: Renouncing ownership will leave the contract without an owner,
478  * thereby removing any functionality that is only available to the owner.
479  */
480  function renounceOwnership() external onlyOwner {
481      _transferOwnership(address(0));
482  }
483
484  /**
485  * @dev Transfers ownership of the contract to a new account (`newOwner`).
486  * Can only be called by the current owner.
487  */
488  function transferOwnership(address newOwner) public onlyOwner {
489      require(newOwner != address(0), "Ownable: new owner is the zero address");
490      _transferOwnership(newOwner);
491  }
492
493  /**
494  * @dev Transfers ownership of the contract to a new account (`newOwner`).
495  * Internal function without access restriction.
496  */
497  function _transferOwnership(address newOwner) internal {
498      address oldOwner = _owner;
499      _owner = newOwner;
500      emit OwnershipTransferred(oldOwner, newOwner);

```

```

501     }
502 }
503
504 /**
505  * @dev Contract module that helps prevent reentrant calls to a function.
506  *
507  * Inheriting from `ReentrancyGuard` will make the {nonReentrant} modifier
508  * available, which can be applied to functions to make sure there are no nested
509  * (reentrant) calls to them.
510  *
511  * Note that because there is a single `nonReentrant` guard, functions marked as
512  * `nonReentrant` may not call one another. This can be worked around by making
513  * those functions `private`, and then adding `external` `nonReentrant` entry
514  * points to them.
515  *
516  * TIP: If you would like to learn more about reentrancy and alternative ways
517  * to protect against it, check out our blog post
518  * https://blog.openzeppelin.com/reentrancy-after-istanbul/ [Reentrancy After Istanbul].
519  */
520 abstract contract ReentrancyGuard {
521     // Booleans are more expensive than uint256 or any type that takes up a full
522     // word because each write operation emits an extra SLOAD to first read the
523     // slot's contents, replace the bits taken up by the boolean, and then write
524     // back. This is the compiler's defense against contract upgrades and
525     // pointer aliasing, and it cannot be disabled.
526
527     // The values being non-zero value makes deployment a bit more expensive,
528     // but in exchange the refund on every call to nonReentrant will be lower in
529     // amount. Since refunds are capped to a percentage of the total
530     // transaction's gas, it is best to keep them low in cases like this one, to
531     // increase the likelihood of the full refund coming into effect.
532     uint256 private constant _NOT_ENTERED = 1;
533     uint256 private constant _ENTERED = 2;
534
535     uint256 private _status;
536
537     constructor() {
538         _status = _NOT_ENTERED;
539     }
540
541     /**
542     * @dev Prevents a contract from calling itself, directly or indirectly.
543     * Calling a `nonReentrant` function from another `nonReentrant`
544     * function is not supported. It is possible to prevent this from happening
545     * by making the `nonReentrant` function external, and making it call a
546     * `private` function that does the actual work.
547     */
548     modifier nonReentrant() {
549         _nonReentrantBefore();
550         _;

```



```

551     _nonReentrantAfter();
552 }
553
554 function _nonReentrantBefore() private {
555     // On the first call to nonReentrant, _notEntered will be true
556     require(_status != _ENTERED, "ReentrancyGuard: reentrant call");
557
558     // Any calls to nonReentrant after this point will fail
559     _status = _ENTERED;
560 }
561
562 function _nonReentrantAfter() private {
563     // By storing the original value once again, a refund is triggered (see
564     // https://eips.ethereum.org/EIPS/eip-2200)
565     _status = _NOT_ENTERED;
566 }
567 }
568
569
570
571 /**
572  * @dev Interface of the ERC20 standard as defined in the EIP.
573  */
574 interface IERC20 {
575     /**
576     * @dev Emitted when `value` tokens are moved from one account (`from`) to
577     * another (`to`).
578     *
579     * Note that `value` may be zero.
580     */
581     event Transfer(address indexed from, address indexed to, uint256 value);
582
583     /**
584     * @dev Emitted when the allowance of a `spender` for an `owner` is set by
585     * a call to {approve}. `value` is the new allowance.
586     */
587     event Approval(address indexed owner, address indexed spender, uint256 value);
588
589     /**
590     * @dev Returns the amount of tokens in existence.
591     */
592     function totalSupply() external view returns (uint256);
593
594     /**
595     * @dev Returns the amount of tokens owned by `account`.
596     */
597     function balanceOf(address account) external view returns (uint256);
598
599     /**
600     * @dev Moves `amount` tokens from the caller's account to `to`.

```

```

601      *
602      * Returns a boolean value indicating whether the operation succeeded.
603      *
604      * Emits a {Transfer} event.
605      */
606      function transfer(address to, uint256 amount) external returns (bool);
607
608      /**
609      * @dev Returns the remaining number of tokens that `spender` will be
610      * allowed to spend on behalf of `owner` through {transferFrom}. This is
611      * zero by default.
612      *
613      * This value changes when {approve} or {transferFrom} are called.
614      */
615      function allowance(address from, address spender) external view returns (uint256);
616
617      /**
618      * @dev Sets `amount` as the allowance of `spender` over the caller's tokens.
619      *
620      * Returns a boolean value indicating whether the operation succeeded.
621      *
622      * IMPORTANT: Beware that changing an allowance with this method brings the risk
623      * that someone may use both the old and the new allowance by unfortunate
624      * transaction ordering. One possible solution to mitigate this race
625      * condition is to first reduce the spender's allowance to 0 and set the
626      * desired value afterwards:
627      * https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
628      *
629      * Emits an {Approval} event.
630      */
631      function approve(address spender, uint256 amount) external returns (bool);
632
633      /**
634      * @dev Moves `amount` tokens from `from` to `to` using the
635      * allowance mechanism. `amount` is then deducted from the caller's
636      * allowance.
637      *
638      * Returns a boolean value indicating whether the operation succeeded.
639      *
640      * Emits a {Transfer} event.
641      */
642      function transferFrom(
643          address from,
644          address to,
645          uint256 amount
646      ) external returns (bool);
647  }
648
649  /// @title IERC1643 Document Management (part of the ERC1400 Security Token Standards)
650  /// @dev See https://github.com/SecurityTokenStandard/EIP-Spec

```

```

651
652 interface IERC1643 {
653
654     // Document Management
655     function getDocument(bytes32 _name) external view returns (string memory, bytes32, uint256);
656     function setDocument(bytes32 _name, string memory _uri, bytes32 _documentHash) external;
657     function removeDocument(bytes32 _name) external;
658     function getAllDocuments() external view returns (bytes32[] memory);
659
660     // Document Events
661     event DocumentRemoved(bytes32 indexed name, string uri, bytes32 documentHash);
662     event DocumentUpdated(bytes32 indexed name, string uri, bytes32 documentHash);
663
664 }
665
666 /**
667  * @title IERC1400 security token standard
668  * @dev See https://github.com/SecurityTokenStandard/EIP-Spec/blob/master/eip/eip-1400.md
669  */
670 interface IERC1400 is IERC20, IERC1643 {
671
672     // ***** Token Information *****
673     function balanceOfByPartition(bytes32 partition, address tokenHolder) external view returns (uint256);
674     function partitionsOf(address tokenHolder) external view returns (bytes32[] memory);
675
676     // ***** Transfers *****
677     function transferWithData(address to, uint256 value, bytes calldata data) external;
678     function transferFromWithData(address from, address to, uint256 value, bytes calldata data) external;
679
680     // ***** Partition Token Transfers *****
681     function transferByPartition(bytes32 partition, address to, uint256 value, bytes calldata data) external returns (bytes32);
682     function operatorTransferByPartition(bytes32 partition, address from, address to, uint256 value, bytes calldata data, bytes calldata operatorData) external returns (bytes32);
683     function allowanceByPartition(bytes32 partition, address owner, address spender) external view returns (uint256);
684
685     // ***** Controller Operation *****
686     function isControllable() external view returns (bool);
687     // function controllerTransfer(address from, address to, uint256 value, bytes calldata data, bytes calldata operatorData) external; // removed because same action can be achieved with "operatorTransferByPartition"
688     // function controllerRedeem(address tokenHolder, uint256 value, bytes calldata data, bytes calldata operatorData) external; // removed because same action can be achieved with "operatorRedeemByPartition"
689
690     // ***** Operator Management *****
691     function authorizeOperator(address operator) external;
692     function revokeOperator(address operator) external;
693     function authorizeOperatorByPartition(bytes32 partition, address operator) external;
694     function revokeOperatorByPartition(bytes32 partition, address operator) external;
695
696     // ***** Operator Information *****
697     function isOperator(address operator, address tokenHolder) external view returns (bool);
698     function isOperatorForPartition(bytes32 partition, address operator, address tokenHolder) external view returns (bool);
699
700     // ***** Token Issuance *****

```



```

701 function isIssuable() external view returns (bool);
702 function issue(address tokenHolder, uint256 value, bytes calldata data) external;
703 function issueByPartition(bytes32 partition, address tokenHolder, uint256 value, bytes calldata data) external;
704
705 // ***** Token Redemption *****
706 function redeem(uint256 value, bytes calldata data) external;
707 function redeemFrom(address tokenHolder, uint256 value, bytes calldata data) external;
708 function redeemByPartition(bytes32 partition, uint256 value, bytes calldata data) external;
709 function operatorRedeemByPartition(bytes32 partition, address tokenHolder, uint256 value, bytes calldata operatorData) external;
710
711 event TransferByPartition(
712     bytes32 indexed fromPartition,
713     address operator,
714     address indexed from,
715     address indexed to,
716     uint256 value,
717     bytes data,
718     bytes operatorData
719 );
720
721 event ChangedPartition(
722     bytes32 indexed fromPartition,
723     bytes32 indexed toPartition,
724     uint256 value
725 );
726
727 // ***** Operator Events *****
728 event AuthorizedOperator(address indexed operator, address indexed tokenHolder);
729 event RevokedOperator(address indexed operator, address indexed tokenHolder);
730 event AuthorizedOperatorByPartition(bytes32 indexed partition, address indexed operator, address indexed tokenHolder);
731 event RevokedOperatorByPartition(bytes32 indexed partition, address indexed operator, address indexed tokenHolder);
732
733 // ***** Issuance / Redemption Events *****
734 event Issued(address indexed operator, address indexed to, uint256 value, bytes data);
735 event Redeemed(address indexed operator, address indexed from, uint256 value, bytes data);
736 event IssuedByPartition(bytes32 indexed partition, address indexed operator, address indexed to, uint256 value, bytes data, bytes operatorData);
737 event RedeemedByPartition(bytes32 indexed partition, address indexed operator, address indexed from, uint256 value, bytes operatorData);
738
739 }
740
741 /**
742  * Reason codes - ERC-1066
743  *
744  * To improve the token holder experience, canTransfer MUST return a reason byte code
745  * on success or failure based on the ERC-1066 application-specific status codes specified below.
746  * An implementation can also return arbitrary data as a bytes32 to provide additional
747  * information not captured by the reason code.
748  *
749  * Code Reason
750  * 0x50 transfer failure

```

751 * 0x51 transfer success
752 * 0x52 insufficient balance
753 * 0x53 insufficient allowance
754 * 0x54 transfers halted (contract paused)
755 * 0x55 funds locked (lockup period)
756 * 0x56 invalid sender
757 * 0x57 invalid receiver
758 * 0x58 invalid operator (transfer agent)
759 * 0x59
760 * 0x5a
761 * 0x5b
762 * 0x5a
763 * 0x5b
764 * 0x5c
765 * 0x5d
766 * 0x5e
767 * 0x5f token meta or info
768 *
769 * These codes are being discussed at: <https://ethereum-magicians.org/t/erc-1066-ethereum-status-codes-esc/283/24>
770 */
771
772
773 /**
774 * @title ERC1400
775 * @dev ERC1400 logic
776 */
777 contract ERC1400 is IERC20, IERC1400, Ownable, ERC1820Client, ERC1820Implementer, MinterRole, DomainAware, ReentrancyGuard {
778
779 // Token
780 string constant internal ERC1400_INTERFACE_NAME = "ERC1400Token";
781 string constant internal ERC20_INTERFACE_NAME = "ERC20Token";
782
783 // Token extensions
784 string constant internal ERC1400_TOKENS_CHECKER = "ERC1400TokensChecker";
785 string constant internal ERC1400_TOKENS_VALIDATOR = "ERC1400TokensValidator";
786
787 // User extensions
788 string constant internal ERC1400_TOKENS_SENDER = "ERC1400TokensSender";
789 string constant internal ERC1400_TOKENS_RECIPIENT = "ERC1400TokensRecipient";
790
791 /***** Token description *****/
792 string internal _name;
793 string internal _symbol;
794 uint256 internal _granularity;
795 uint256 internal _totalSupply;
796 bool internal _migrated;
797 /*****/
798
799
800 /***** Token behaviours *****/

```

801 // Indicate whether the token can still be controlled by operators or not anymore.
802 bool internal _isControllable;
803
804 // Indicate whether the token can still be issued by the issuer or not anymore.
805 bool internal _isIssuable;
806 /*****
807
808
809 /***** ERC20 Token mappings *****/
810 // Mapping from tokenHolder to balance.
811 mapping(address => uint256) internal _balances;
812
813 // Mapping from (tokenHolder, spender) to allowed value.
814 mapping (address => mapping (address => uint256)) internal _allowed;
815 /*****
816
817
818 /***** Documents *****/
819 struct Doc {
820     string docURI;
821     bytes32 docHash;
822     uint256 timestamp;
823 }
824 // Mapping for documents.
825 mapping(bytes32 => Doc) internal _documents;
826 mapping(bytes32 => uint256) internal _indexOfDocHashes;
827 bytes32[] internal _docHashes;
828 /*****
829
830
831 /***** Partitions mappings *****/
832 // List of partitions.
833 bytes32[] internal _totalPartitions;
834
835 // Mapping from partition to their index.
836 mapping (bytes32 => uint256) internal _indexOfTotalPartitions;
837
838 // Mapping from partition to global balance of corresponding partition.
839 mapping (bytes32 => uint256) internal _totalSupplyByPartition;
840
841 // Mapping from tokenHolder to their partitions.
842 mapping (address => bytes32[]) internal _partitionsOf;
843
844 // Mapping from (tokenHolder, partition) to their index.
845 mapping (address => mapping (bytes32 => uint256)) internal _indexOfPartitionsOf;
846
847 // Mapping from (tokenHolder, partition) to balance of corresponding partition.
848 mapping (address => mapping (bytes32 => uint256)) internal _balanceOfByPartition;
849
850 // List of token default partitions (for ERC20 compatibility).

```



```

851 bytes32[] internal _defaultPartitions;
852 /*****
853
854
855 /***** Global operators mappings *****/
856 // Mapping from (operator, tokenHolder) to authorized status. [TOKEN-HOLDER-SPECIFIC]
857 mapping(address => mapping(address => bool)) internal _authorizedOperator;
858
859 // Array of controllers. [GLOBAL - NOT TOKEN-HOLDER-SPECIFIC]
860 address[] internal _controllers;
861
862 // Mapping from operator to controller status. [GLOBAL - NOT TOKEN-HOLDER-SPECIFIC]
863 mapping(address => bool) internal _isController;
864 /*****
865
866
867 /***** Partition operators mappings *****/
868 // Mapping from (partition, tokenHolder, spender) to allowed value. [TOKEN-HOLDER-SPECIFIC]
869 mapping(bytes32 => mapping (address => mapping (address => uint256))) internal _allowedByPartition;
870
871 // Mapping from (tokenHolder, partition, operator) to 'approved for partition' status. [TOKEN-HOLDER-SPECIFIC]
872 mapping (address => mapping (bytes32 => mapping (address => bool))) internal _authorizedOperatorByPartition;
873
874 // Mapping from partition to controllers for the partition. [NOT TOKEN-HOLDER-SPECIFIC]
875 mapping (bytes32 => address[]) internal _controllersByPartition;
876
877 // Mapping from (partition, operator) to PartitionController status. [NOT TOKEN-HOLDER-SPECIFIC]
878 mapping (bytes32 => mapping (address => bool)) internal _isControllerByPartition;
879 /*****
880
881
882 /***** Modifiers *****/
883 /**
884  * @dev Modifier to verify if token is issuable.
885 */
886 modifier isIssuableToken() {
887     require(!_isIssuable, "55"); // 0x55 funds locked (lockup period)
888     _;
889 }
890 /**
891  * @dev Modifier to make a function callable only when the contract is not migrated.
892 */
893 modifier isNotMigratedToken() {
894     require(!_migrated, "54"); // 0x54 transfers halted (contract paused)
895     _;
896 }
897 /**
898  * @dev Modifier to verify if sender is a minter.
899 */
900 modifier onlyMinter() override {

```

```

901     require(isMinter(msg.sender) || owner() == _msgSender());
902     _;
903 }
904 /*****
905
906
907 /***** Events (additional - not mandatory) *****/
908 event ApprovalByPartition(bytes32 indexed partition, address indexed owner, address indexed spender, uint256 value);
909 /*****
910
911
912 /**
913  * @dev Initialize ERC1400 + register the contract implementation in ERC1820Registry.
914  * @param name_ Name of the token.
915  * @param symbol_ Symbol of the token.
916  * @param granularity_ Granularity of the token.
917  * @param defaultPartitions_ Partitions chosen by default, when partition is
918  * not specified, like the case ERC20 tranfers.
919  */
920 constructor(
921     string memory name_,
922     string memory symbol_,
923     uint256 granularity_,
924     bytes32[] memory defaultPartitions_
925 )
926 {
927     _name = name_;
928     _symbol = symbol_;
929     _totalSupply = 0;
930     require(granularity_ >= 1); // Constructor Blocked - Token granularity can not be lower than 1
931     _granularity = granularity_;
932
933     _defaultPartitions = defaultPartitions_;
934
935     _isControllable = true;
936     _isIssuable = true;
937
938     // Register contract in ERC1820 registry
939     ERC1820Client.setInterfaceImplementation(ERC1400_INTERFACE_NAME, address(this));
940     ERC1820Client.setInterfaceImplementation(ERC20_INTERFACE_NAME, address(this));
941
942     // Indicate token verifies ERC1400 and ERC20 interfaces
943     ERC1820Implementer._setInterface(ERC1400_INTERFACE_NAME); // For migration
944     ERC1820Implementer._setInterface(ERC20_INTERFACE_NAME); // For migration
945 }
946
947
948 /*****
949 /***** EXTERNAL FUNCTIONS (ERC20 INTERFACE) *****/
950 /*****

```

```

951
952
953  /**
954   * @dev Get the total number of issued tokens.
955   * @return Total supply of tokens currently in circulation.
956   */
957   function totalSupply() external override view returns (uint256) {
958       return _totalSupply;
959   }
960  /**
961   * @dev Get the balance of the account with address 'tokenHolder'.
962   * @param tokenHolder Address for which the balance is returned.
963   * @return Amount of token held by 'tokenHolder' in the token contract.
964   */
965   function balanceOf(address tokenHolder) external override view returns (uint256) {
966       return _balances[tokenHolder];
967   }
968  /**
969   * @dev Transfer token for a specified address.
970   * @param to The address to transfer to.
971   * @param value The value to be transferred.
972   * @return A boolean that indicates if the operation was successful.
973   */
974   function transfer(address to, uint256 value) external override returns (bool) {
975       _transferByDefaultPartitions(msg.sender, msg.sender, to, value, "");
976       return true;
977   }
978  /**
979   * @dev Check the value of tokens that an owner allowed to a spender.
980   * @param from address The address which owns the funds.
981   * @param spender address The address which will spend the funds.
982   * @return A uint256 specifying the value of tokens still available for the spender.
983   */
984   function allowance(address from, address spender) external override view returns (uint256) {
985       return _allowed[from][spender];
986   }
987  /**
988   * @dev Approve the passed address to spend the specified amount of tokens on behalf of 'msg.sender'.
989   * @param spender The address which will spend the funds.
990   * @param value The amount of tokens to be spent.
991   * @return A boolean that indicates if the operation was successful.
992   */
993   function approve(address spender, uint256 value) external override returns (bool) {
994       require(spender != address(0), "56"); // 0x56 invalid sender
995       _allowed[msg.sender][spender] = value;
996       emit Approval(msg.sender, spender, value);
997       return true;
998   }
999  /**
1000   * @dev Transfer tokens from one address to another.

```



```

1001 * @param from The address which you want to transfer tokens from.
1002 * @param to The address which you want to transfer to.
1003 * @param value The amount of tokens to be transferred.
1004 * @return A boolean that indicates if the operation was successful.
1005 */
1006 function transferFrom(address from, address to, uint256 value) external override returns (bool) {
1007     require( _isOperator(msg.sender, from)
1008         || (value <= _allowed[from][msg.sender]), "53"); // 0x53 insufficient allowance
1009
1010     if(_allowed[from][msg.sender] >= value) {
1011         _allowed[from][msg.sender] = _allowed[from][msg.sender] - value;
1012     } else {
1013         _allowed[from][msg.sender] = 0;
1014     }
1015
1016     _transferByDefaultPartitions(msg.sender, from, to, value, "");
1017     return true;
1018 }
1019
1020
1021 /*****
1022 /***** EXTERNAL FUNCTIONS (ERC1400 INTERFACE) *****/
1023 /*****/
1024
1025
1026 /***** Document Management *****/
1027 /**
1028 * @dev Access a document associated with the token.
1029 * @param shortName Short name (represented as a bytes32) associated to the document.
1030 * @return Requested document + document hash + document timestamp.
1031 */
1032 function getDocument(bytes32 shortName) external override view returns (string memory, bytes32, uint256) {
1033     require(bytes(_documents[shortName].docURI).length != 0); // Action Blocked - Empty document
1034     return (
1035         _documents[shortName].docURI,
1036         _documents[shortName].docHash,
1037         _documents[shortName].timestamp
1038     );
1039 }
1040 /**
1041 * @dev Associate a document with the token.
1042 * @param shortName Short name (represented as a bytes32) associated to the document.
1043 * @param uri Document content.
1044 * @param documentHash Hash of the document [optional parameter].
1045 */
1046 function setDocument(bytes32 shortName, string calldata uri, bytes32 documentHash) external override {
1047     require(_isController(msg.sender));
1048     _documents[shortName] = Doc({
1049         docURI: uri,
1050         docHash: documentHash,

```

```

1051         timestamp: block.timestamp
1052     });
1053
1054     if (_indexOfDocHashes[documentHash] == 0) {
1055         _docHashes.push(documentHash);
1056         _indexOfDocHashes[documentHash] = _docHashes.length;
1057     }
1058
1059     emit DocumentUpdated(shortName, uri, documentHash);
1060 }
1061
1062 function removeDocument(bytes32 shortName) external override {
1063     require(!_isController[msg.sender], "Unauthorized");
1064     require(bytes(_documents[shortName].docURI).length != 0, "Document doesnt exist"); // Action Blocked - Empty document
1065
1066     Doc memory data = _documents[shortName];
1067
1068     uint256 index1 = _indexOfDocHashes[data.docHash];
1069     require(index1 > 0, "Invalid index"); //Indexing starts at 1, 0 is not allowed
1070
1071     // move the last item into the index being vacated
1072     bytes32 lastValue = _docHashes[_docHashes.length - 1];
1073     _docHashes[index1 - 1] = lastValue; // adjust for 1-based indexing
1074     _indexOfDocHashes[lastValue] = index1;
1075
1076     //_totalPartitions.length -= 1;
1077     _docHashes.pop();
1078     _indexOfDocHashes[data.docHash] = 0;
1079
1080     delete _documents[shortName];
1081
1082     emit DocumentRemoved(shortName, data.docURI, data.docHash);
1083 }
1084
1085 function getAllDocuments() external override view returns (bytes32[] memory) {
1086     return _docHashes;
1087 }
1088 /*****
1089
1090
1091 /***** Token Information *****/
1092 /**
1093  * @dev Get balance of a tokenholder for a specific partition.
1094  * @param partition Name of the partition.
1095  * @param tokenHolder Address for which the balance is returned.
1096  * @return Amount of token of partition 'partition' held by 'tokenHolder' in the token contract.
1097  */
1098 function balanceOfByPartition(bytes32 partition, address tokenHolder) external override view returns (uint256) {
1099     return _balanceOfByPartition[tokenHolder][partition];
1100 }

```

```

1101  /**
1102   * @dev Get partitions index of a tokenholder.
1103   * @param tokenHolder Address for which the partitions index are returned.
1104   * @return Array of partitions index of 'tokenHolder'.
1105   */
1106   function partitionsOf(address tokenHolder) external override view returns (bytes32[] memory) {
1107       return _partitionsOf[tokenHolder];
1108   }
1109   /*****
1110
1111
1112   /***** Transfers *****/
1113   /**
1114   * @dev Transfer the amount of tokens from the address 'msg.sender' to the address 'to'.
1115   * @param to Token recipient.
1116   * @param value Number of tokens to transfer.
1117   * @param data Information attached to the transfer, by the token holder.
1118   */
1119   function transferWithData(address to, uint256 value, bytes calldata data) external override {
1120       _transferByDefaultPartitions(msg.sender, msg.sender, to, value, data);
1121   }
1122   /**
1123   * @dev Transfer the amount of tokens on behalf of the address 'from' to the address 'to'.
1124   * @param from Token holder (or 'address(0)' to set from to 'msg.sender').
1125   * @param to Token recipient.
1126   * @param value Number of tokens to transfer.
1127   * @param data Information attached to the transfer, and intended for the token holder ('from').
1128   */
1129   function transferFromWithData(address from, address to, uint256 value, bytes calldata data) external override virtual {
1130       require( _isOperator(msg.sender, from)
1131           || (value <= _allowed[from][msg.sender]), "53"); // 0x53 insufficient allowance
1132
1133       if(_allowed[from][msg.sender] >= value) {
1134           _allowed[from][msg.sender] = _allowed[from][msg.sender] - value;
1135       } else {
1136           _allowed[from][msg.sender] = 0;
1137       }
1138
1139       _transferByDefaultPartitions(msg.sender, from, to, value, data);
1140   }
1141   /*****
1142
1143
1144   /***** Partition Token Transfers *****/
1145   /**
1146   * @dev Transfer tokens from a specific partition.
1147   * @param partition Name of the partition.
1148   * @param to Token recipient.
1149   * @param value Number of tokens to transfer.
1150   * @param data Information attached to the transfer, by the token holder.

```



```

1151 * @return Destination partition.
1152 */
1153 function transferByPartition(
1154     bytes32 partition,
1155     address to,
1156     uint256 value,
1157     bytes calldata data
1158 )
1159 external
1160 override
1161 nonReentrant
1162 returns (bytes32)
1163 {
1164     {
1165         return _transferByPartition(partition, msg.sender, msg.sender, to, value, data, "");
1166     }
1167 }
1168 /**
1169  * @dev Transfer tokens from a specific partition through an operator.
1170  * @param partition Name of the partition.
1171  * @param from Token holder.
1172  * @param to Token recipient.
1173  * @param value Number of tokens to transfer.
1174  * @param data Information attached to the transfer. [CAN CONTAIN THE DESTINATION PARTITION]
1175  * @param operatorData Information attached to the transfer, by the operator.
1176  * @return Destination partition.
1177 */
1178 function operatorTransferByPartition(
1179     bytes32 partition,
1180     address from,
1181     address to,
1182     uint256 value,
1183     bytes calldata data,
1184     bytes calldata operatorData
1185 )
1186 external
1187 override
1188 returns (bytes32)
1189 {
1190     //We want to check if the msg.sender is an authorized operator for `from`
1191     //(msg.sender == from OR msg.sender is authorized by from OR msg.sender is a controller if this token is controlable)
1192     //OR
1193     //We want to check if msg.sender is an `allowed` operator/spender for `from`
1194     require(_isOperatorForPartition(partition, msg.sender, from)
1195         || (value <= _allowedByPartition[partition][from][msg.sender]), "53"); // 0x53 insufficient allowance
1196
1197     if(_allowedByPartition[partition][from][msg.sender] >= value) {
1198         _allowedByPartition[partition][from][msg.sender] = _allowedByPartition[partition][from][msg.sender] - value;
1199     } else {
1200         _allowedByPartition[partition][from][msg.sender] = 0;

```

```

1201     }
1202
1203     return _transferByPartition(partition, msg.sender, from, to, value, data, operatorData);
1204 }
1205 /*****
1206
1207
1208 /***** Controller Operation *****/
1209 /**
1210  * @dev Know if the token can be controlled by operators.
1211  * If a token returns 'false' for 'isControllable()' then it MUST always return 'false' in the future.
1212  * @return bool 'true' if the token can still be controlled by operators, 'false' if it can't anymore.
1213  */
1214 function isControllable() external override view returns (bool) {
1215     return _isControllable;
1216 }
1217 /*****
1218
1219
1220 /***** Operator Management *****/
1221 /**
1222  * @dev Set a third party operator address as an operator of 'msg.sender' to transfer
1223  * and redeem tokens on its behalf.
1224  * @param operator Address to set as an operator for 'msg.sender'.
1225  */
1226 function authorizeOperator(address operator) external override {
1227     require(operator != msg.sender);
1228     _authorizedOperator[operator][msg.sender] = true;
1229     emit AuthorizedOperator(operator, msg.sender);
1230 }
1231 /**
1232  * @dev Remove the right of the operator address to be an operator for 'msg.sender'
1233  * and to transfer and redeem tokens on its behalf.
1234  * @param operator Address to rescind as an operator for 'msg.sender'.
1235  */
1236 function revokeOperator(address operator) external override {
1237     require(operator != msg.sender);
1238     _authorizedOperator[operator][msg.sender] = false;
1239     emit RevokedOperator(operator, msg.sender);
1240 }
1241 /**
1242  * @dev Set 'operator' as an operator for 'msg.sender' for a given partition.
1243  * @param partition Name of the partition.
1244  * @param operator Address to set as an operator for 'msg.sender'.
1245  */
1246 function authorizeOperatorByPartition(bytes32 partition, address operator) external override {
1247     _authorizedOperatorByPartition[msg.sender][partition][operator] = true;
1248     emit AuthorizedOperatorByPartition(partition, operator, msg.sender);
1249 }
1250 /**

```

```

1251 * @dev Remove the right of the operator address to be an operator on a given
1252 * partition for 'msg.sender' and to transfer and redeem tokens on its behalf.
1253 * @param partition Name of the partition.
1254 * @param operator Address to rescind as an operator on given partition for 'msg.sender'.
1255 */
1256 function revokeOperatorByPartition(bytes32 partition, address operator) external override {
1257     _authorizedOperatorByPartition[msg.sender][partition][operator] = false;
1258     emit RevokedOperatorByPartition(partition, operator, msg.sender);
1259 }
1260 /*****
1261
1262
1263 /***** Operator Information *****/
1264 /**
1265 * @dev Indicate whether the operator address is an operator of the tokenHolder address.
1266 * @param operator Address which may be an operator of tokenHolder.
1267 * @param tokenHolder Address of a token holder which may have the operator address as an operator.
1268 * @return 'true' if operator is an operator of 'tokenHolder' and 'false' otherwise.
1269 */
1270 function isOperator(address operator, address tokenHolder) external override view returns (bool) {
1271     return _isOperator(operator, tokenHolder);
1272 }
1273 /**
1274 * @dev Indicate whether the operator address is an operator of the tokenHolder
1275 * address for the given partition.
1276 * @param partition Name of the partition.
1277 * @param operator Address which may be an operator of tokenHolder for the given partition.
1278 * @param tokenHolder Address of a token holder which may have the operator address as an operator for the given partition.
1279 * @return 'true' if 'operator' is an operator of 'tokenHolder' for partition 'partition' and 'false' otherwise.
1280 */
1281 function isOperatorForPartition(bytes32 partition, address operator, address tokenHolder) external override view returns (bool) {
1282     return _isOperatorForPartition(partition, operator, tokenHolder);
1283 }
1284 /*****
1285
1286
1287 /***** Token Issuance *****/
1288 /**
1289 * @dev Know if new tokens can be issued in the future.
1290 * @return bool 'true' if tokens can still be issued by the issuer, 'false' if they can't anymore.
1291 */
1292 function isIssuable() external override view returns (bool) {
1293     return _isIssuable;
1294 }
1295 /**
1296 * @dev Issue tokens from default partition.
1297 * @param tokenHolder Address for which we want to issue tokens.
1298 * @param value Number of tokens issued.
1299 * @param data Information attached to the issuance, by the issuer.
1300 */

```



```

1301 function issue(address tokenHolder, uint256 value, bytes calldata data)
1302 external
1303 override
1304 onlyMinter
1305 nonReentrant
1306 issuableToken
1307 {
1308     require(_defaultPartitions.length != 0, "55"); // 0x55 funds locked (lockup period)
1309
1310     _issueByPartition(_defaultPartitions[0], msg.sender, tokenHolder, value, data);
1311 }
1312 /**
1313  * @dev Issue tokens from a specific partition.
1314  * @param partition Name of the partition.
1315  * @param tokenHolder Address for which we want to issue tokens.
1316  * @param value Number of tokens issued.
1317  * @param data Information attached to the issuance, by the issuer.
1318  */
1319 function issueByPartition(bytes32 partition, address tokenHolder, uint256 value, bytes calldata data)
1320 external
1321 override
1322 onlyMinter
1323 nonReentrant
1324 issuableToken
1325 {
1326     _issueByPartition(partition, msg.sender, tokenHolder, value, data);
1327 }
1328 /*****
1329
1330
1331  /***** Token Redemption *****/
1332  /**
1333   * @dev Redeem the amount of tokens from the address 'msg.sender'.
1334   * @param value Number of tokens to redeem.
1335   * @param data Information attached to the redemption, by the token holder.
1336   */
1337 function redeem(uint256 value, bytes calldata data)
1338 external
1339 override
1340 nonReentrant
1341 {
1342     _redeemByDefaultPartitions(msg.sender, msg.sender, value, data);
1343 }
1344 /**
1345  * @dev Redeem the amount of tokens on behalf of the address from.
1346  * @param from Token holder whose tokens will be redeemed (or address(0) to set from to msg.sender).
1347  * @param value Number of tokens to redeem.
1348  * @param data Information attached to the redemption.
1349  */
1350 function redeemFrom(address from, uint256 value, bytes calldata data)

```



```

1351     external
1352     override
1353     virtual
1354     nonReentrant
1355     {
1356         require(!_isOperator(msg.sender, from)
1357             || (value <= _allowed[from][msg.sender]), "53"); // 0x53 insufficient allowance
1358
1359         if(_allowed[from][msg.sender] >= value) {
1360             _allowed[from][msg.sender] = _allowed[from][msg.sender] - value;
1361         } else {
1362             _allowed[from][msg.sender] = 0;
1363         }
1364
1365         _redeemByDefaultPartitions(msg.sender, from, value, data);
1366     }
1367     /**
1368      * @dev Redeem tokens of a specific partition.
1369      * @param partition Name of the partition.
1370      * @param value Number of tokens redeemed.
1371      * @param data Information attached to the redemption, by the redeemer.
1372      */
1373     function redeemByPartition(bytes32 partition, uint256 value, bytes calldata data)
1374     external
1375     override
1376     nonReentrant
1377     {
1378         _redeemByPartition(partition, msg.sender, msg.sender, value, data, "");
1379     }
1380     /**
1381      * @dev Redeem tokens of a specific partition.
1382      * @param partition Name of the partition.
1383      * @param tokenHolder Address for which we want to redeem tokens.
1384      * @param value Number of tokens redeemed
1385      * @param operatorData Information attached to the redemption, by the operator.
1386      */
1387     function operatorRedeemByPartition(bytes32 partition, address tokenHolder, uint256 value, bytes calldata operatorData)
1388     external
1389     override
1390     nonReentrant
1391     nonReentrant
1392     {
1393         require(!_isOperatorForPartition(partition, msg.sender, tokenHolder) || value <= _allowedByPartition[partition][tokenHolder][msg.sender], "58"); // 0x58 invalid operator (transfer agent)
1394
1395         if(_allowedByPartition[partition][tokenHolder][msg.sender] >= value) {
1396             _allowedByPartition[partition][tokenHolder][msg.sender] = _allowedByPartition[partition][tokenHolder][msg.sender] - value;
1397         } else {
1398             _allowedByPartition[partition][tokenHolder][msg.sender] = 0;
1399         }
1400

```

```

1401     _redeemByPartition(partition, msg.sender, tokenHolder, value, "", operatorData);
1402 }
1403 /*****
1404
1405
1406 /*****/
1407 /***** EXTERNAL FUNCTIONS (ADDITIONAL - NOT MANDATORY) *****/
1408 /*****/
1409
1410
1411 /***** Token description *****/
1412 /**
1413  * @dev Get the name of the token, e.g., "MyToken".
1414  * @return Name of the token.
1415  */
1416 function name() external view returns(string memory) {
1417     return _name;
1418 }
1419 /**
1420  * @dev Get the symbol of the token, e.g., "MYT".
1421  * @return Symbol of the token.
1422  */
1423 function symbol() external view returns(string memory) {
1424     return _symbol;
1425 }
1426 /**
1427  * @dev Get the number of decimals of the token.
1428  * @return The number of decimals of the token. For retrocompatibility, decimals are forced to 18 in ERC1400.
1429  */
1430 function decimals() external pure returns(uint8) {
1431     return uint8(18);
1432 }
1433 /**
1434  * @dev Get the smallest part of the token that's not divisible.
1435  * @return The smallest non-divisible part of the token.
1436  */
1437 function granularity() external view returns(uint256) {
1438     return _granularity;
1439 }
1440 /**
1441  * @dev Get list of existing partitions.
1442  * @return Array of all existing partitions.
1443  */
1444 function totalPartitions() external view returns (bytes32[] memory) {
1445     return _totalPartitions;
1446 }
1447 /**
1448  * @dev Get the total number of issued tokens for a given partition.
1449  * @param partition Name of the partition.
1450  * @return Total supply of tokens currently in circulation, for a given partition.

```

```

1451 */
1452 function totalSupplyByPartition(bytes32 partition) external view returns (uint256) {
1453     return _totalSupplyByPartition[partition];
1454 }
1455 /*****
1456
1457
1458 /***** Token behaviours *****/
1459 /**
1460  * @dev Definitely renounce the possibility to control tokens on behalf of tokenHolders.
1461  * Once set to false, '_isControllable' can never be set to 'true' again.
1462  */
1463 function renounceControl() external onlyOwner {
1464     _isControllable = false;
1465 }
1466 /**
1467  * @dev Definitely renounce the possibility to issue new tokens.
1468  * Once set to false, '_isIssuable' can never be set to 'true' again.
1469  */
1470 function renounceIssuance() external onlyOwner {
1471     _isIssuable = false;
1472 }
1473 /*****
1474
1475
1476 /***** Token controllers *****/
1477 /**
1478  * @dev Get the list of controllers as defined by the token contract.
1479  * @return List of addresses of all the controllers.
1480  */
1481 function controllers() external view returns (address[] memory) {
1482     return _controllers;
1483 }
1484 /**
1485  * @dev Get controllers for a given partition.
1486  * @param partition Name of the partition.
1487  * @return Array of controllers for partition.
1488  */
1489 function controllersByPartition(bytes32 partition) external view returns (address[] memory) {
1490     return _controllersByPartition[partition];
1491 }
1492 /**
1493  * @dev Set list of token controllers.
1494  * @param operators Controller addresses.
1495  */
1496 function setControllers(address[] calldata operators) external onlyOwner {
1497     _setControllers(operators);
1498 }
1499 /**
1500  * @dev Set list of token partition controllers.

```



```

1501 * @param partition Name of the partition.
1502 * @param operators Controller addresses.
1503 */
1504 function setPartitionControllers(bytes32 partition, address[] calldata operators) external onlyOwner {
1505     _setPartitionControllers(partition, operators);
1506 }
1507 /*****
1508
1509
1510 /***** Token default partitions *****/
1511 /**
1512 * @dev Get default partitions to transfer from.
1513 * Function used for ERC20 retrocompatibility.
1514 * For example, a security token may return the bytes32("unrestricted").
1515 * @return Array of default partitions.
1516 */
1517 function getDefaultPartitions() external view returns (bytes32[] memory) {
1518     return _defaultPartitions;
1519 }
1520 /**
1521 * @dev Set default partitions to transfer from.
1522 * Function used for ERC20 retrocompatibility.
1523 * @param partitions partitions to use by default when not specified.
1524 */
1525 function setDefaultPartitions(bytes32[] calldata partitions) external onlyOwner {
1526     _defaultPartitions = partitions;
1527 }
1528 /*****
1529
1530
1531 /***** Partition Token Allowances *****/
1532 /**
1533 * @dev Check the value of tokens that an owner allowed to a spender.
1534 * @param partition Name of the partition.
1535 * @param owner address The address which owns the funds.
1536 * @param spender address The address which will spend the funds.
1537 * @return A uint256 specifying the value of tokens still available for the spender.
1538 */
1539 function allowanceByPartition(bytes32 partition, address owner, address spender) external override view returns (uint256) {
1540     return _allowedByPartition[partition][owner][spender];
1541 }
1542 /**
1543 * @dev Approve the passed address to spend the specified amount of tokens on behalf of 'msg.sender'.
1544 * @param partition Name of the partition.
1545 * @param spender The address which will spend the funds.
1546 * @param value The amount of tokens to be spent.
1547 * @return A boolean that indicates if the operation was successful.
1548 */
1549 function approveByPartition(bytes32 partition, address spender, uint256 value) external returns (bool) {
1550     require(spender != address(0), "56"); // 0x56 invalid sender

```



```

1551     _allowedByPartition[partition][msg.sender][spender] = value;
1552     emit ApprovalByPartition(partition, msg.sender, spender, value);
1553     return true;
1554 }
1555 /*****/
1556
1557
1558 /***** Token extension *****/
1559 /**
1560  * @dev Set token extension contract address.
1561  * The extension contract can for example verify "ERC1400TokensValidator" or "ERC1400TokensChecker" interfaces.
1562  * If the extension is an "ERC1400TokensValidator", it will be called everytime a transfer is executed.
1563  * @param extension Address of the extension contract.
1564  * @param interfaceLabel Interface label of extension contract.
1565  * @param removeOldExtensionRoles If set to 'true', the roles of the old extension(minter, controller) will be removed extension.
1566  * @param addMinterRoleForExtension If set to 'true', the extension contract will be added as minter.
1567  * @param addControllerRoleForExtension If set to 'true', the extension contract will be added as controller.
1568 */
1569 function setTokenExtension(address extension, string calldata interfaceLabel, bool removeOldExtensionRoles, bool addMinterRoleForExtension, bool addControllerRoleForExtension) external onlyOwner {
1570     _setTokenExtension(extension, interfaceLabel, removeOldExtensionRoles, addMinterRoleForExtension, addControllerRoleForExtension);
1571 }
1572 /*****/
1573
1574 /***** Token migration *****/
1575 /**
1576  * @dev Migrate contract.
1577  *
1578  * ==> CAUTION: DEFINITIVE ACTION
1579  *
1580  * This function shall be called once a new version of the smart contract has been created.
1581  * Once this function is called:
1582  * - The address of the new smart contract is set in ERC1820 registry
1583  * - If the choice is definitive, the current smart contract is turned off and can never be used again
1584  *
1585  * @param newContractAddress Address of the new version of the smart contract.
1586  * @param definitive If set to 'true' the contract is turned off definitely.
1587 */
1588 function migrate(address newContractAddress, bool definitive) external onlyOwner {
1589     _migrate(newContractAddress, definitive);
1590 }
1591 /*****/
1592
1593
1594 /*****/
1595 /***** INTERNAL FUNCTIONS *****/
1596 /*****/
1597
1598
1599 /***** Token Transfers *****/
1600 /**

```

```

1601     * @dev Perform the transfer of tokens.
1602     * @param from Token holder.
1603     * @param to Token recipient.
1604     * @param value Number of tokens to transfer.
1605     */
1606     function _transferWithData(
1607         address from,
1608         address to,
1609         uint256 value
1610     )
1611     internal
1612     isNotMigratedToken
1613     {
1614         require(_isMultiple(value), "50"); // 0x50 transfer failure
1615         require(to != address(0), "57"); // 0x57 invalid receiver
1616         require(_balances[from] >= value, "52"); // 0x52 insufficient balance
1617
1618         _balances[from] = _balances[from] - value;
1619         _balances[to] = _balances[to] + value;
1620
1621         emit Transfer(from, to, value); // ERC20 retrocompatibility
1622     }
1623     /**
1624     * @dev Transfer tokens from a specific partition.
1625     * @param fromPartition Partition of the tokens to transfer.
1626     * @param operator The address performing the transfer.
1627     * @param from Token holder.
1628     * @param to Token recipient.
1629     * @param value Number of tokens to transfer.
1630     * @param data Information attached to the transfer. [CAN CONTAIN THE DESTINATION PARTITION]
1631     * @param operatorData Information attached to the transfer, by the operator (if any).
1632     * @return Destination partition.
1633     */
1634     function _transferByPartition(
1635         bytes32 fromPartition,
1636         address operator,
1637         address from,
1638         address to,
1639         uint256 value,
1640         bytes memory data,
1641         bytes memory operatorData
1642     )
1643     internal
1644     returns (bytes32)
1645     {
1646         require(_balanceOfByPartition[from][fromPartition] >= value, "52"); // 0x52 insufficient balance
1647
1648         bytes32 toPartition = fromPartition;
1649
1650         if(operatorData.length != 0 && data.length >= 64) {

```

```

1651     toPartition = _getDestinationPartition(fromPartition, data);
1652 }
1653
1654 _callSenderExtension(fromPartition, operator, from, to, value, data, operatorData);
1655 _callTokenExtension(fromPartition, operator, from, to, value, data, operatorData);
1656
1657 _removeTokenFromPartition(from, fromPartition, value);
1658 _transferWithData(from, to, value);
1659 _addTokenToPartition(to, toPartition, value);
1660
1661 _callRecipientExtension(toPartition, operator, from, to, value, data, operatorData);
1662
1663 emit TransferByPartition(fromPartition, operator, from, to, value, data, operatorData);
1664
1665 if(toPartition != fromPartition) {
1666     emit ChangedPartition(fromPartition, toPartition, value);
1667 }
1668
1669 return toPartition;
1670 }
1671 /**
1672  * @dev Transfer tokens from default partitions.
1673  * Function used for ERC20 retrocompatibility.
1674  * @param operator The address performing the transfer.
1675  * @param from Token holder.
1676  * @param to Token recipient.
1677  * @param value Number of tokens to transfer.
1678  * @param data Information attached to the transfer, and intended for the token holder ('from') [CAN CONTAIN THE DESTINATION PARTITION].
1679  */
1680 function _transferByDefaultPartitions(
1681     address operator,
1682     address from,
1683     address to,
1684     uint256 value,
1685     bytes memory data
1686 )
1687 internal
1688 {
1689     require(_defaultPartitions.length != 0, "55"); // 0x55 funds locked (lockup period)
1690
1691     uint256 _remainingValue = value;
1692     uint256 _localBalance;
1693
1694     for (uint i = 0; i < _defaultPartitions.length; i++) {
1695         _localBalance = _balanceOfByPartition[from][_defaultPartitions[i]];
1696         if(_remainingValue <= _localBalance) {
1697             _transferByPartition(_defaultPartitions[i], operator, from, to, _remainingValue, data, "");
1698             _remainingValue = 0;
1699             break;
1700         } else if (_localBalance != 0) {

```



```

1701     _transferByPartition(_defaultPartitions[i], operator, from, to, _localBalance, data, "");
1702     _remainingValue = _remainingValue - _localBalance;
1703 }
1704 }
1705
1706     require(_remainingValue == 0, "52"); // 0x52 insufficient balance
1707 }
1708 /**
1709  * @dev Retrieve the destination partition from the 'data' field.
1710  * By convention, a partition change is requested ONLY when 'data' starts
1711  * with the flag: 0xffffffffffffffffffffffffffffffffffffffff
1712  * When the flag is detected, the destination tranche is extracted from the
1713  * 32 bytes following the flag.
1714  * @param fromPartition Partition of the tokens to transfer.
1715  * @param data Information attached to the transfer. [CAN CONTAIN THE DESTINATION PARTITION]
1716  * @return toPartition Destination partition.
1717 */
1718 function _getDestinationPartition(bytes32 fromPartition, bytes memory data) internal pure returns(bytes32 toPartition) {
1719     bytes32 changePartitionFlag = 0xffffffffffffffffffffffffffffffffffffffff;
1720     bytes32 flag;
1721     assembly {
1722         flag := mload(add(data, 32))
1723     }
1724     if(flag == changePartitionFlag) {
1725         assembly {
1726             toPartition := mload(add(data, 64))
1727         }
1728     } else {
1729         toPartition = fromPartition;
1730     }
1731 }
1732 /**
1733  * @dev Remove a token from a specific partition.
1734  * @param from Token holder.
1735  * @param partition Name of the partition.
1736  * @param value Number of tokens to transfer.
1737 */
1738 function _removeTokenFromPartition(address from, bytes32 partition, uint256 value) internal {
1739     _balanceOfByPartition[from][partition] = _balanceOfByPartition[from][partition] - value;
1740     _totalSupplyByPartition[partition] = _totalSupplyByPartition[partition] - value;
1741
1742     // If the total supply is zero, finds and deletes the partition.
1743     if(_totalSupplyByPartition[partition] == 0) {
1744         uint256 index1 = _indexOfTotalPartitions[partition];
1745         require(index1 > 0, "50"); // 0x50 transfer failure
1746
1747         // move the last item into the index being vacated
1748         bytes32 lastValue = _totalPartitions[_totalPartitions.length - 1];
1749         _totalPartitions[index1 - 1] = lastValue; // adjust for 1-based indexing
1750         _indexOfTotalPartitions[lastValue] = index1;

```



```

1751
1752     //_totalPartitions.length -= 1;
1753     _totalPartitions.pop();
1754     _indexOfTotalPartitions[partition] = 0;
1755 }
1756
1757 // If the balance of the TokenHolder's partition is zero, finds and deletes the partition.
1758 if(_balanceOfByPartition[from][partition] == 0) {
1759     uint256 index2 = _indexOfPartitionsOf[from][partition];
1760     require(index2 > 0, "50"); // 0x50 transfer failure
1761
1762     // move the last item into the index being vacated
1763     bytes32 lastValue = _partitionsOf[from][_partitionsOf[from].length - 1];
1764     _partitionsOf[from][index2 - 1] = lastValue; // adjust for 1-based indexing
1765     _indexOfPartitionsOf[from][lastValue] = index2;
1766
1767     //_partitionsOf[from].length -= 1;
1768     _partitionsOf[from].pop();
1769     _indexOfPartitionsOf[from][partition] = 0;
1770 }
1771 }
1772 /**
1773  * @dev Add a token to a specific partition.
1774  * @param to Token recipient.
1775  * @param partition Name of the partition.
1776  * @param value Number of tokens to transfer.
1777  */
1778 function _addTokenToPartition(address to, bytes32 partition, uint256 value) internal {
1779     if(value != 0) {
1780         if (_indexOfPartitionsOf[to][partition] == 0) {
1781             _partitionsOf[to].push(partition);
1782             _indexOfPartitionsOf[to][partition] = _partitionsOf[to].length;
1783         }
1784         _balanceOfByPartition[to][partition] = _balanceOfByPartition[to][partition] + value;
1785
1786         if (_indexOfTotalPartitions[partition] == 0) {
1787             _totalPartitions.push(partition);
1788             _indexOfTotalPartitions[partition] = _totalPartitions.length;
1789         }
1790         _totalSupplyByPartition[partition] = _totalSupplyByPartition[partition] + value;
1791     }
1792 }
1793 /**
1794  * @dev Check if 'value' is multiple of the granularity.
1795  * @param value The quantity that want's to be checked.
1796  * @return 'true' if 'value' is a multiple of the granularity.
1797  */
1798 function _isMultiple(uint256 value) internal view returns(bool) {
1799     return(uint256(value / _granularity) * _granularity == value);
1800 }

```

```

1801 /*****
1802
1803
1804 /***** Hooks *****/
1805 /**
1806  * @dev Check for 'ERC1400TokensSender' user extension in ERC1820 registry and call it.
1807  * @param partition Name of the partition (bytes32 to be left empty for transfers where partition is not specified).
1808  * @param operator Address which triggered the balance decrease (through transfer or redemption).
1809  * @param from Token holder.
1810  * @param to Token recipient for a transfer and 0x for a redemption.
1811  * @param value Number of tokens the token holder balance is decreased by.
1812  * @param data Extra information.
1813  * @param operatorData Extra information, attached by the operator (if any).
1814 */
1815 function _callSenderExtension(
1816     bytes32 partition,
1817     address operator,
1818     address from,
1819     address to,
1820     uint256 value,
1821     bytes memory data,
1822     bytes memory operatorData
1823 )
1824 internal
1825 {
1826     address senderImplementation;
1827     senderImplementation = interfaceAddr(from, ERC1400_TOKENS_SENDER);
1828     if (senderImplementation != address(0)) {
1829         IERC1400TokensSender(senderImplementation).tokensToTransfer(msg.data, partition, operator, from, to, value, data, operatorData);
1830     }
1831 }
1832 /**
1833  * @dev Check for 'ERC1400TokensValidator' token extension in ERC1820 registry and call it.
1834  * @param partition Name of the partition (bytes32 to be left empty for transfers where partition is not specified).
1835  * @param operator Address which triggered the balance decrease (through transfer or redemption).
1836  * @param from Token holder.
1837  * @param to Token recipient for a transfer and 0x for a redemption.
1838  * @param value Number of tokens the token holder balance is decreased by.
1839  * @param data Extra information.
1840  * @param operatorData Extra information, attached by the operator (if any).
1841 */
1842 function _callTokenExtension(
1843     bytes32 partition,
1844     address operator,
1845     address from,
1846     address to,
1847     uint256 value,
1848     bytes memory data,
1849     bytes memory operatorData
1850 )

```

```

1851 internal
1852 {
1853     address validatorImplementation;
1854     validatorImplementation = interfaceAddr(address(this), ERC1400_TOKENS_VALIDATOR);
1855     if (validatorImplementation != address(0)) {
1856         IERC1400TokensValidator(validatorImplementation).tokensToValidate(msg.data, partition, operator, from, to, value, data, operatorData);
1857     }
1858 }
1859 /**
1860  * @dev Check for 'ERC1400TokensRecipient' user extension in ERC1820 registry and call it.
1861  * @param partition Name of the partition (bytes32 to be left empty for transfers where partition is not specified).
1862  * @param operator Address which triggered the balance increase (through transfer or issuance).
1863  * @param from Token holder for a transfer and 0x for an issuance.
1864  * @param to Token recipient.
1865  * @param value Number of tokens the recipient balance is increased by.
1866  * @param data Extra information, intended for the token holder ('from').
1867  * @param operatorData Extra information attached by the operator (if any).
1868 */
1869 function _callRecipientExtension(
1870     bytes32 partition,
1871     address operator,
1872     address from,
1873     address to,
1874     uint256 value,
1875     bytes memory data,
1876     bytes memory operatorData
1877 )
1878 internal
1879 virtual
1880 {
1881     address recipientImplementation;
1882     recipientImplementation = interfaceAddr(to, ERC1400_TOKENS_RECIPIENT);
1883
1884     if (recipientImplementation != address(0)) {
1885         IERC1400TokensRecipient(recipientImplementation).tokensReceived(msg.data, partition, operator, from, to, value, data, operatorData);
1886     }
1887 }
1888 /*****
1889
1890
1891  /***** Operator Information *****/
1892  /**
1893   * @dev Indicate whether the operator address is an operator of the tokenHolder address.
1894   * @param operator Address which may be an operator of 'tokenHolder'.
1895   * @param tokenHolder Address of a token holder which may have the 'operator' address as an operator.
1896   * @return 'true' if 'operator' is an operator of 'tokenHolder' and 'false' otherwise.
1897   */
1898   function _isOperator(address operator, address tokenHolder) internal view returns (bool) {
1899       return (operator == tokenHolder
1900           || _authorizedOperator[operator][tokenHolder]

```



```

1901         || (_isControllable && _isController[operator])
1902     );
1903 }
1904 /**
1905  * @dev Indicate whether the operator address is an operator of the tokenHolder
1906  * address for the given partition.
1907  * @param partition Name of the partition.
1908  * @param operator Address which may be an operator of tokenHolder for the given partition.
1909  * @param tokenHolder Address of a token holder which may have the operator address as an operator for the given partition.
1910  * @return 'true' if 'operator' is an operator of 'tokenHolder' for partition 'partition' and 'false' otherwise.
1911  */
1912 function _isOperatorForPartition(bytes32 partition, address operator, address tokenHolder) internal view returns (bool) {
1913     return (_isOperator(operator, tokenHolder)
1914         || _authorizedOperatorByPartition[tokenHolder][partition][operator]
1915         || (_isControllable && _isControllerByPartition[partition][operator])
1916     );
1917 }
1918 /*****
1919
1920
1921  /***** Token Issuance *****/
1922  /**
1923   * @dev Perform the issuance of tokens.
1924   * @param operator Address which triggered the issuance.
1925   * @param to Token recipient.
1926   * @param value Number of tokens issued.
1927   * @param data Information attached to the issuance, and intended for the recipient (to).
1928   */
1929 function _issue(address operator, address to, uint256 value, bytes memory data)
1930 internal
1931 isNotMigratedToken
1932 {
1933     require(_isMultiple(value), "50"); // 0x50 transfer failure
1934     require(to != address(0), "57"); // 0x57 invalid receiver
1935
1936     _totalSupply = _totalSupply + value;
1937     _balances[to] = _balances[to] + value;
1938
1939     emit Issued(operator, to, value, data);
1940     emit Transfer(address(0), to, value); // ERC20 retrocompatibility
1941 }
1942 /**
1943  * @dev Issue tokens from a specific partition.
1944  * @param toPartition Name of the partition.
1945  * @param operator The address performing the issuance.
1946  * @param to Token recipient.
1947  * @param value Number of tokens to issue.
1948  * @param data Information attached to the issuance.
1949  */
1950 function _issueByPartition(

```



```

1951     bytes32 toPartition,
1952     address operator,
1953     address to,
1954     uint256 value,
1955     bytes memory data
1956 )
1957 internal
1958 {
1959     _callTokenExtension(toPartition, operator, address(0), to, value, data, "");
1960
1961     _issue(operator, to, value, data);
1962     _addTokenToPartition(to, toPartition, value);
1963
1964     _callRecipientExtension(toPartition, operator, address(0), to, value, data, "");
1965
1966     emit IssuedByPartition(toPartition, operator, to, value, data, "");
1967 }
1968 /*****
1969
1970
1971 /***** Token Redemption *****/
1972 /**
1973  * @dev Perform the token redemption.
1974  * @param operator The address performing the redemption.
1975  * @param from Token holder whose tokens will be redeemed.
1976  * @param value Number of tokens to redeem.
1977  * @param data Information attached to the redemption.
1978 */
1979 function _redeem(address operator, address from, uint256 value, bytes memory data)
1980 internal
1981 isNotMigratedToken
1982 {
1983     require(_isMultiple(value), "50"); // 0x50 transfer failure
1984     require(from != address(0), "56"); // 0x56 invalid sender
1985     require(_balances[from] >= value, "52"); // 0x52 insufficient balance
1986
1987     _balances[from] = _balances[from] - value;
1988     _totalSupply = _totalSupply - value;
1989
1990     emit Redeemed(operator, from, value, data);
1991     emit Transfer(from, address(0), value); // ERC20 retrocompatibility
1992 }
1993 /**
1994  * @dev Redeem tokens of a specific partition.
1995  * @param fromPartition Name of the partition.
1996  * @param operator The address performing the redemption.
1997  * @param from Token holder whose tokens will be redeemed.
1998  * @param value Number of tokens to redeem.
1999  * @param data Information attached to the redemption.
2000  * @param operatorData Information attached to the redemption, by the operator (if any).

```

```

2001 */
2002 function _redeemByPartition(
2003     bytes32 fromPartition,
2004     address operator,
2005     address from,
2006     uint256 value,
2007     bytes memory data,
2008     bytes memory operatorData
2009 )
2010 internal
2011 {
2012     require(_balanceOfByPartition[from][fromPartition] >= value, "52"); // 0x52 insufficient balance
2013
2014     _callSenderExtension(fromPartition, operator, from, address(0), value, data, operatorData);
2015     _callTokenExtension(fromPartition, operator, from, address(0), value, data, operatorData);
2016
2017     _removeTokenFromPartition(from, fromPartition, value);
2018     _redeem(operator, from, value, data);
2019
2020     emit RedeemedByPartition(fromPartition, operator, from, value, operatorData);
2021 }
2022 /**
2023  * @dev Redeem tokens from a default partitions.
2024  * @param operator The address performing the redeem.
2025  * @param from Token holder.
2026  * @param value Number of tokens to redeem.
2027  * @param data Information attached to the redemption.
2028  */
2029 function _redeemByDefaultPartitions(
2030     address operator,
2031     address from,
2032     uint256 value,
2033     bytes memory data
2034 )
2035 internal
2036 {
2037     require(_defaultPartitions.length != 0, "55"); // 0x55 funds locked (lockup period)
2038
2039     uint256 _remainingValue = value;
2040     uint256 _localBalance;
2041
2042     for (uint i = 0; i < _defaultPartitions.length; i++) {
2043         _localBalance = _balanceOfByPartition[from][_defaultPartitions[i]];
2044         if(_remainingValue <= _localBalance) {
2045             _redeemByPartition(_defaultPartitions[i], operator, from, _remainingValue, data, "");
2046             _remainingValue = 0;
2047             break;
2048         } else {
2049             _redeemByPartition(_defaultPartitions[i], operator, from, _localBalance, data, "");
2050             _remainingValue = _remainingValue - _localBalance;

```

```

2051     }
2052 }
2053
2054     require(_remainingValue == 0, "52"); // 0x52  insufficient balance
2055 }
2056 /*****/
2057
2058
2059 /***** Transfer Validity *****/
2060 /**
2061  * @dev Know the reason on success or failure based on the EIP-1066 application-specific status codes.
2062  * @param payload Payload of the initial transaction.
2063  * @param partition Name of the partition.
2064  * @param operator The address performing the transfer.
2065  * @param from Token holder.
2066  * @param to Token recipient.
2067  * @param value Number of tokens to transfer.
2068  * @param data Information attached to the transfer. [CAN CONTAIN THE DESTINATION PARTITION]
2069  * @param operatorData Information attached to the transfer, by the operator (if any).
2070  * @return ESC (Ethereum Status Code) following the EIP-1066 standard.
2071  * @return Additional bytes32 parameter that can be used to define
2072  * application specific reason codes with additional details (for example the
2073  * transfer restriction rule responsible for making the transfer operation invalid).
2074  * @return Destination partition.
2075 */
2076 function _canTransfer(bytes memory payload, bytes32 partition, address operator, address from, address to, uint256 value, bytes memory data, bytes memory operatorData)
2077 internal
2078 view
2079 returns (bytes1, bytes32, bytes32)
2080 {
2081     address checksImplementation = interfaceAddr(address(this), ERC1400_TOKENS_CHECKER);
2082
2083     if((checksImplementation != address(0))) {
2084         return IERC1400TokensChecker(checksImplementation).canTransferByPartition(payload, partition, operator, from, to, value, data, operatorData);
2085     }
2086     else {
2087         return(hex"00", "", partition);
2088     }
2089 }
2090 /*****/
2091
2092
2093 /*****/
2094 /***** INTERNAL FUNCTIONS (ADDITIONAL - NOT MANDATORY) *****/
2095 /*****/
2096
2097
2098 /***** Token controllers *****/
2099 /**
2100  * @dev Set list of token controllers.

```



```

2101 * @param operators Controller addresses.
2102 */
2103 function _setControllers(address[] memory operators) internal {
2104     for (uint i = 0; i<_controllers.length; i++){
2105         _isController[_controllers[i]] = false;
2106     }
2107     for (uint j = 0; j<operators.length; j++){
2108         _isController[operators[j]] = true;
2109     }
2110     _controllers = operators;
2111 }
2112 /**
2113  * @dev Set list of token partition controllers.
2114  * @param partition Name of the partition.
2115  * @param operators Controller addresses.
2116  */
2117 function _setPartitionControllers(bytes32 partition, address[] memory operators) internal {
2118     for (uint i = 0; i<_controllersByPartition[partition].length; i++){
2119         _isControllerByPartition[partition][_controllersByPartition[partition][i]] = false;
2120     }
2121     for (uint j = 0; j<operators.length; j++){
2122         _isControllerByPartition[partition][operators[j]] = true;
2123     }
2124     _controllersByPartition[partition] = operators;
2125 }
2126 /*****
2127
2128
2129 /***** Token extension *****/
2130 /**
2131  * @dev Set token extension contract address.
2132  * The extension contract can for example verify "ERC1400TokensValidator" or "ERC1400TokensChecker" interfaces.
2133  * If the extension is an "ERC1400TokensValidator", it will be called everytime a transfer is executed.
2134  * @param extension Address of the extension contract.
2135  * @param interfaceLabel Interface label of extension contract.
2136  * @param removeOldExtensionRoles If set to 'true', the roles of the old extension(minter, controller) will be removed extension.
2137  * @param addMinterRoleForExtension If set to 'true', the extension contract will be added as minter.
2138  * @param addControllerRoleForExtension If set to 'true', the extension contract will be added as controller.
2139  */
2140 function _setTokenExtension(address extension, string memory interfaceLabel, bool removeOldExtensionRoles, bool addMinterRoleForExtension, bool addControllerRoleForExtension) internal {
2141     address oldExtension = interfaceAddr(address(this), interfaceLabel);
2142
2143     if (oldExtension != address(0) && removeOldExtensionRoles) {
2144         if(isMinter(oldExtension)) {
2145             _removeMinter(oldExtension);
2146         }
2147         _isController[oldExtension] = false;
2148     }
2149
2150     ERC1820Client.setInterfaceImplementation(interfaceLabel, extension);

```



```

2151     if(addMinterRoleForExtension && !isMinter(extension)) {
2152         _addMinter(extension);
2153     }
2154     if (addControllerRoleForExtension) {
2155         _isController[extension] = true;
2156     }
2157 }
2158 /*****
2159
2160
2161 /***** Token migration *****/
2162 /**
2163  * @dev Migrate contract.
2164  *
2165  * ==> CAUTION: DEFINITIVE ACTION
2166  *
2167  * This function shall be called once a new version of the smart contract has been created.
2168  * Once this function is called:
2169  * - The address of the new smart contract is set in ERC1820 registry
2170  * - If the choice is definitive, the current smart contract is turned off and can never be used again
2171  *
2172  * @param newContractAddress Address of the new version of the smart contract.
2173  * @param definitive If set to 'true' the contract is turned off definitely.
2174  */
2175 function _migrate(address newContractAddress, bool definitive) internal {
2176     ERC1820Client.setInterfaceImplementation(ERC20_INTERFACE_NAME, newContractAddress);
2177     ERC1820Client.setInterfaceImplementation(ERC1400_INTERFACE_NAME, newContractAddress);
2178     if(definitive) {
2179         _migrated = true;
2180     }
2181 }
2182 /*****
2183
2184 /***** Domain Aware *****/
2185 function domainName() public override view returns (string memory) {
2186     return _name;
2187 }
2188
2189 function domainVersion() public override pure returns (string memory) {
2190     return "1";
2191 }
2192 /*****
2193 }
2194
2195 /**
2196  * @notice Interface to the extension types
2197  */
2198 interface IExtensionTypes {
2199     enum CertificateValidation {
2200         None,

```

```

2201     NonceBased,
2202     SaltBased
2203 }
2204 }
2205
2206 /**
2207  * @notice Interface to the extension contract
2208  */
2209 abstract contract Extension is IExtensionTypes {
2210     function registerTokenSetup(
2211         address token,
2212         CertificateValidation certificateActivated,
2213         bool allowlistActivated,
2214         bool blocklistActivated,
2215         bool granularityByPartitionActivated,
2216         bool holdsActivated,
2217         address[] calldata operators
2218     ) external virtual;
2219
2220     function addCertificateSigner(
2221         address token,
2222         address account
2223     ) external virtual;
2224 }
2225
2226 contract MTC is ERC1400, IExtensionTypes {
2227
2228     uint256 public constant ISSUED_TOKENS = 21_000 * 10 ** 18;
2229     uint256 public constant FOR_SALE_TOKENS = 21_000 * 10 ** 18;
2230
2231     bytes32 public constant MTC_DEFAULT_PARTITION = keccak256(abi.encodePacked("MTC_DEFAULT_PARTITION"));
2232
2233     bytes32[] private MTC_DEFAULT_PARTITIONS = [bytes32(MTC_DEFAULT_PARTITION)];
2234     uint256 constant private GRANULARITY = 10 ** 18;
2235
2236     /**
2237      * @dev Initialize ERC1400.
2238      * @param newOwner Address whom contract ownership shall be transferred to.
2239      * @param seller Address who will sell tokens.
2240      */
2241     constructor(
2242         address newOwner,
2243         address seller
2244     )
2245     ERC1400("Monetec Token", "MTC", GRANULARITY, MTC_DEFAULT_PARTITIONS)
2246     {
2247         require(newOwner != address(0), "MTC: wrong address");
2248         require(seller != address(0), "MTC: wrong address");
2249
2250         transferOwnership(newOwner);

```

```

2251
2252     _issueByPartition(MTC_DEFAULT_PARTITION, msg.sender, newOwner, ISSUED_TOKENS, "");
2253     _issuable = false;
2254
2255     _transferByDefaultPartitions(msg.sender, newOwner, seller, FOR_SALE_TOKENS, "");
2256 }
2257
2258 /**
2259  * @dev Initialize extension.
2260  * @param extension Address of token extension.
2261  * @param certificateSigner Address of the off-chain service which signs the
2262  * conditional ownership certificates required for token transfers, issuance,
2263  * redemption (Cf. CertificateController.sol).
2264  * @param certificateActivated If set to 'true', the certificate controller
2265  * is activated at contract creation.
2266  * @param controllers_ Array of initial controllers.
2267  */
2268 function initExtention(
2269     address extension,
2270     address certificateSigner,
2271     CertificateValidation certificateActivated,
2272     address[] memory controllers_
2273 ) external onlyOwner
2274 {
2275     if(extension != address(0)) {
2276         Extension(extension).registerTokenSetup(
2277             address(this), // token
2278             certificateActivated, // certificateActivated
2279             false, // allowlistActivated
2280             false, // blocklistActivated
2281             true, // granularityByPartitionActivated
2282             true, // holdsActivated
2283             controllers_ // token controllers
2284         );
2285
2286         if(certificateSigner != address(0)) {
2287             Extension(extension).addCertificateSigner(address(this), certificateSigner);
2288         }
2289
2290         _setTokenExtension(extension, ERC1400_TOKENS_VALIDATOR, true, true, true);
2291     }
2292     _setControllers(controllers_);
2293 }
2294
2295
2296
2297 /***** Transfer Validity *****/
2298 /**
2299  * @dev Know the reason on success or failure based on the EIP-1066 application-specific status codes.
2300  * @param partition Name of the partition.

```



```

2301 * @param to Token recipient.
2302 * @param value Number of tokens to transfer.
2303 * @param data Information attached to the transfer, by the token holder. [CONTAINS THE CONDITIONAL OWNERSHIP CERTIFICATE]
2304 * @return ESC (Ethereum Status Code) following the EIP-1066 standard.
2305 * @return Additional bytes32 parameter that can be used to define
2306 * application specific reason codes with additional details (for example the
2307 * transfer restriction rule responsible for making the transfer operation invalid).
2308 * @return Destination partition.
2309 */
2310 function canTransferByPartition(bytes32 partition, address to, uint256 value, bytes calldata data)
2311 external
2312 view
2313 returns (bytes1, bytes32, bytes32)
2314 {
2315     return ERC1400._canTransfer(
2316         _replaceFunctionSelector(this.transferByPartition.selector, msg.data), // 0xf3d490db: 4 first bytes of keccak256(transferByPartition(bytes32,address,uint256,bytes))
2317         partition,
2318         msg.sender,
2319         msg.sender,
2320         to,
2321         value,
2322         data,
2323         ""
2324     );
2325 }
2326 /**
2327  * @dev Know the reason on success or failure based on the EIP-1066 application-specific status codes.
2328  * @param partition Name of the partition.
2329  * @param from Token holder.
2330  * @param to Token recipient.
2331  * @param value Number of tokens to transfer.
2332  * @param data Information attached to the transfer. [CAN CONTAIN THE DESTINATION PARTITION]
2333  * @param operatorData Information attached to the transfer, by the operator. [CONTAINS THE CONDITIONAL OWNERSHIP CERTIFICATE]
2334  * @return ESC (Ethereum Status Code) following the EIP-1066 standard.
2335  * @return Additional bytes32 parameter that can be used to define
2336  * application specific reason codes with additional details (for example the
2337  * transfer restriction rule responsible for making the transfer operation invalid).
2338  * @return Destination partition.
2339  */
2340 function canOperatorTransferByPartition(bytes32 partition, address from, address to, uint256 value, bytes calldata data, bytes calldata operatorData)
2341 external
2342 view
2343 returns (bytes1, bytes32, bytes32)
2344 {
2345     return ERC1400._canTransfer(
2346         _replaceFunctionSelector(this.operatorTransferByPartition.selector, msg.data), // 0x8c0dee9c: 4 first bytes of keccak256(operatorTransferByPartition(bytes32,address,address,uint256,bytes,bytes))
2347         partition,
2348         msg.sender,
2349         from,
2350         to,

```



```

2351     value,
2352     data,
2353     operatorData
2354 );
2355 }
2356 /**
2357  * @dev Replace function selector
2358  * @param functionSig Replacement function selector.
2359  * @param payload Payload, where function selector needs to be replaced.
2360  */
2361 function _replaceFunctionSelector(bytes4 functionSig, bytes memory payload) internal pure returns(bytes memory) {
2362     bytes memory updatedPayload = new bytes(payload.length);
2363     for (uint i = 0; i<4; i++){
2364         updatedPayload[i] = functionSig[i];
2365     }
2366     for (uint j = 4; j<payload.length; j++){
2367         updatedPayload[j] = payload[j];
2368     }
2369     return updatedPayload;
2370 }
2371 /*****
2372
2373     */

```