# USDTieredSTO

- **Introduced in:** 2.0.0
- **Contract name:** USDTieredSTO.sol
- **Type:** STO Module
- **Compatible Protocol Version:** TBD
- **Associated LucidChart:** TBD

## How it works

Allows a security token to be issued in return for investment (security token offering) in various currencies (ETH, POLY & a USD stable coin). The price of tokens is denominated in USD and the STO allows multiple tiers with different price points to be defined. Discounts for investments made in POLY can also be defined.

## Key functionalities (as defined in the Smart Contract)

### Initialization

The contract is initialized with all of the parameters needed to setup the STO (tier information, wallet addresses and so on).

```
    /**
     * @notice Function used to intialize the contract variabl
es
     * @param _startTime Unix timestamp at which offering get
started
     * @param _endTime Unix timestamp at which offering get en
ded
     * @param _ratePerTier Rate (in USD) per tier (* 10**18)
     * @param _tokensPerTierTotal Tokens available in each tie
r
     * @param _nonAccreditedLimitUSD Limit in USD (* 10**18) f
or non-accredited investors
```

```
     * @param _minimumInvestmentUSD Minimun investment in USD (* 10**18)
     * @param _fundRaiseTypes Types of currency used to collect the funds
     * @param _wallet Ethereum account address to hold the funds
     * @param _reserveWallet Ethereum account address to receive unsold tokens
     * @param _usdToken Contract addresses of the stable coin
     */
    function configure(
        uint256 _startTime,
        uint256 _endTime,
        uint256[] _ratePerTier,
        uint256[] _ratePerTierDiscountPoly,
        uint256[] _tokensPerTierTotal,
        uint256[] _tokensPerTierDiscountPoly,
        uint256 _nonAccreditedLimitUSD,
        uint256 _minimumInvestmentUSD,
        FundRaiseType[] _fundRaiseTypes,
        address _wallet,
        address _reserveWallet,
        address _usdToken
    )
```

Configuration can be considered in 5 separate sections, each of which can be modified anytime between the creation of the USDTieredSTO and the start time of the STO. When the issuer is configuring their STO, they need to provide an array of addresses of stable coins they will accept (changed from earlier as the only option was the DAI stable coin).

## Modify Times

Allows you to modify the start or end time of the STO.

```
/**
 * @dev Modifies STO start and end times
 * @param _startTime start time of sto
 * @param _endTime end time of sto
 */
function modifyTimes(
    uint256 _startTime,
    uint256 _endTime
)
```

## Modify Tiers

This function allows you to modify the details of your STOs tiers. Primarily, it allows you to customize the rates you want to set for each tier, the discount rates per tier for investors using POLY, the total amount of tokens you want to allocate to each tier and the token amount of tokens you want to allocate to investors getting a discount because they are buying with POLY.

**A few things to note:**
1. Firstly, you can only use this function if the STO hasn't started.
2. You can't have 0 tokens allocated to a tier
3. Rates in a tier need to be equal to the amount of tokens in tier (You don't want a mismatch between discount rates and tokens per tier)
4. Similarly, the rates per tier for discounted POLY needs to be equal to the total amount of tokens allocated to that tier (You don't want a mismatch between discount tokens per tier and the tokens per tier)
5. The discounted tokens (for investors using Poly) per tier have to be less than or equal to the total tokens per tier
6. The discount rate (for investors using Poly) per tier needs to be less than or equal to the rate per tier

```
/**
 * @dev Modifies STO tiers
 * @param _ratePerTier rate per tier
 * @param _ratePerTierDiscountPoly tier discount rate for inve
sting with Poly
 * @param _tokensPerTierTotal total amount of tokens per tier
 * @param _tokensPerTierDiscountPoly total amount of tokens wi
th a discount for Poly
 */

function modifyTiers(
    uint256[] _ratePerTier,
    uint256[] _ratePerTierDiscountPoly,
    uint256[] _tokensPerTierTotal,
    uint256[] _tokensPerTierDiscountPoly

)
```

## Modify Addresses

Allows you to modify the addresses used for the STO.

```
/**
 * @dev Modifies the needed addresses for the STO.
 * @param _wallet general STO wallet address
 * @param _reserveWallet wallet address for the reserve wallet
 * @param _usdToken wallet address for the USD backed token
 */

 function modifyAddresses(
    address _wallet,
    address _reserveWallet,
    address _usdToken
)
```

## Modify Funding

This function is used to modify the type of funding your STO will be accept as well as general STO structure. This includes adding tiers with discount rates to your STO and more. First, you can only modify the type of funding if the STO has yet to begin. If that is the case, the options that you can alter or add are the following:

1. Fundraise type (in ETH, POLY, DAI or all)
2. The number of tiers you want to have during your STO
3. The amount of tokens you are allocating to each tier
4. The amount of tokens allocated for a discount in your tiers when investors use POLY to invest

```
/**
 * @dev Modifies the type of STO fudning.
 * @param _fundRaiseTypes funding raising types for the STO
 */

 function modifyFunding(
 FundRaiseType[] _fundRaiseTypes


)
```

## Modify Limits

Allows you to modify the limits for non-accredited investors and minimum investment (in USD) for the STO.

```
/**
```

```
 * @dev Modifies the limits for nonaccredited and min investme
nt (USD)
 * @param _nonAccreditedLimitUSD update the the limit for nona
ccredited
 * @param _minimumInvestmentUSD update the the limit for min i
nvestment
 */

function modifyLimits(
uint256 _nonAccreditedLimitUSD,
uint256 _minimumInvestmentUSD

)
```

## Tiers

Each tier defines the following attributes:

```
struct Tier {
    // How many token units a buyer gets per USD in this tier
(multiplied by 10**18)
    uint256 rate;
    // How many token units a buyer gets per USD in this tier
(multiplied by 10**18) when investing in POLY up to tokensDisc
ountPoly
    uint256 rateDiscountPoly;
    // How many tokens are available in this tier (relative to
totalSupply)
    uint256 tokenTotal;
    // How many token units are available in this tier (relati
ve to totalSupply) at the ratePerTierDiscountPoly rate
    uint256 tokensDiscountPoly;
    // How many tokens have been minted in this tier (relative
to totalSupply)
    uint256 mintedTotal;
    // How many tokens have been minted in this tier (relative
to totalSupply) for each fund raise type
    mapping (uint8 => uint256) minted;
    // How many tokens have been minted in this tier (relative
to totalSupply) at discounted POLY rate
    uint256 mintedDiscountPoly;
}
```

You can have as many tiers as needed, and each tier will be filled sequentially. If `tokensDiscountPoly` is not 0, then the tier has some quota of tokens which are offered at a discounted price (`ratePerTierDiscountPoly`) for POLY investments.

N.B.: While theoretically it's possible to have a large number of tiers, due to several loops within the contract's functions there's a limit to how many should be entered. In practice it's unlikely any issuer would have more than 5-7 of them.  We'll force a max of 5 on the dApp.

## Oracles

In order to convert between ETH & POLY investments and USD (in which the STO is denominated) the STO makes use of pricing oracles.

```
/**
 * @dev returns current conversion rate of funds
 * @param _fundRaiseType Fund raise type to get rate of
 */
function getRate(FundRaiseType _fundRaiseType) public view ret
urns (uint256) {
    if (_fundRaiseType == FundRaiseType.ETH) {
        return IOracle(_getOracle(bytes32("ETH"), bytes32("US
D"))).getPrice();
    } else if (_fundRaiseType == FundRaiseType.POLY) {
        return IOracle(_getOracle(bytes32("POLY"), bytes32("US
D"))).getPrice();
    } else if (_fundRaiseType == FundRaiseType.DAI) {
        return 1 * 10**18;
    } else {
        revert("Incorrect funding");
    }
}
```

The `getRate` function allows a caller to determine the current rate for each funding currency.

NB - this rate varies over time, so there is no guarantee that the returned rate will still be valid on a subsequent transaction (see Buying Tokens for how to mitigate this).

## Buying Tokens

Tokens can be issued in return for investments made in ETH, POLY or DAI (or generally any USD stable coin). The currencies accepted are configured in the STO via the `modifyFunding` function.

For each currency type, there are two functions that can be used to make an investment. For example, for ETH we have:

```
function buyWithETH(address _beneficiary) external payable {
    buyWithETHRateLimited(_beneficiary, 0);
}


/**
 * @notice Purchase tokens using ETH
 * @param _beneficiary Address where security tokens will be s
ent
 * @param _minTokens Minumum number of tokens to buy or else r
evert
*/
function buyWithETHRateLimited(address _beneficiary, uint256 _
minTokens) public payable validETH
```

`buyWithETH` accepts ETH as an investment currency and will purchase an amount of tokens that corresponds to the amount of ETH sent with the function.

**2.0.1 UPGRADE:** buyWithETHRateLimited` accepts ETH as an investment currency and will purchase an amount of tokens that corresponds to the amount of ETH sent with the function. However it will also ensure that at least `_minTokens` are purchased or otherwise revert and return the invested ETH. This allows the investor to establish a guarantee on a minimum ETH / USD rate and tier point that their purchase is made.

For investments made in POLY or with a stable coin we have equivalent functions. Before calling these `buyWith…` functions though, the investor must have approved a corresponding amount of tokens to the STO address (called `approve` on the POLY or stable coin contract with the address of the STO contract, and the amount of POLY / stable coin being invested). These functions also take as a parameter the exact amount of tokens being invested.

```
function buyWithPOLY(address _beneficiary, uint256 _investedPO
LY) external;


function buyWithUSD(address _beneficiary, uint256 _investedDA
I) external;


/**
  * @notice Purchase tokens using POLY
  * @param _beneficiary Address where security tokens will be
sent
```

```
  * @param _investedPOLY Amount of POLY invested
  * @param _minTokens Minumum number of tokens to buy or else
revert
  */
function buyWithPOLYRateLimited(address _beneficiary, uint256
_investedPOLY, uint256 _minTokens) public validPOLY;


/**
  * @notice Purchase tokens using DAI
  * @param _beneficiary Address where security tokens will be
sent
  * @param _investedDAI Amount of DAI invested
  * @param _minTokens Minumum number of tokens to buy or else
revert
  */
function buyWithUSDRateLimited(address _beneficiary, uint256 _
investedDAI, uint256 _minTokens) public validDAI;
```

## Refunds

If funds are sent which can't be fully invested (for example if a non-accredited investor limit is reached, or the STO has sold all tokens) any uninvested funds are refunded to the investor.

**2.0.1 UPGRADE:** This can also happen if the underlying security token only allows tokens with a certain granularity and the sent funds would result in purchasing tokens with a more granular quantity.

## Accredited vs. Non-Accredited Investor Limits

**Note:** The below functions work in a way that controls situations when an investor (accredited or not) is buying tokens from an issuer. For example, when an investor wants to invest in a token, these functions control situations such as if there is a minimum limit of investment that every investor must respect or when a non-accredited investor can’t buy anymore tokens due to a non-accredited investor limit. The difference between accredited and non accredited in this situation is that an accredited investor doesn’t have any limit on what they can buy whereas a non accredited investor has a limit (However, accredited may have a minimum amount they have to buy).

## Accredited

Allows you to make investors accredited.

```
/**
@dev Modifies the list of accredited addresses
@param _investors Array of investor addresses to modify
@param _accredited Array of bools specifying accreditation sta
tus
*/


 function changeAccredited(
 address[] _investors,
 bool[] _accredited


)
```

## Non-accredited Limit

Allows you to change the non-accredited investors limit.

```
/**
* @dev Modifies the list of overrides for non-accredited limit
s in USD
* @param _investors Array of investor addresses to modify
* @param _nonAccreditedLimit Array of uints specifying non-acc
redited limits
*/


function changeNonAccreditedLimit(
    address[] _investors,
    uint256[] _nonAccreditedLimit
)
```

## Finalization

The USDTieredSTO can be finalized at any time. Finalizing the STO will close it to any further investments, and mint any remaining unsold tokens to the `reserveWallet` specified in `modifyAddresses`.

```
/**
 * @notice Finalizes the STO and mint remaining tokens to rese
rve address
 * @notice Reserve address must be whitelisted to successfully
finalize
 */
function finalize() public onlyOwner;
```

N.B. When the end of the STO is reached the STO will not auto-mint outstanding tokens. The Issuer has to manually call finalize() in order for the unsold tokens to be minted.

## STO Information (getFunctions)

## Get Rate

**Summary:** This function checks to see if the fundraise type is equal to the fundraise rates in Poly, ETH and DAI. For example, it checks the rates with the oracle to see if the fund raise type equals the fund raise type in Poly, ETH and/or DAI and then the Oracle returns the results. The function reverts if it doesn’t have the correct funding.

```

function getRate(
FundRaiseType _fundRaiseType


)
```

## Get Token Sold

**Summary:** This function allows you to check the total number of tokens that have been sold to investors.

```
/**
* @notice Return the total no. of tokens sold
* @return uint256 Total number of tokens sold


*/
function getTokensSold() public view returns (uint256) {
if (isFinalized)
return totalTokensSold;
else
return getTokensMinted();
}
```

## Get Tokens Minted

**Summary:** This function allows you to check the total number of tokens that have been minted.

```
/**
* @notice Return the total no. of tokens minted
* @return uint256 Total number of tokens minted


*/
```

```
function getTokensMinted() public view returns (uint256) {
uint256 tokensMinted;
for (uint8 i = 0; i < mintedPerTierTotal.length; i++) {
tokensMinted = tokensMinted.add(mintedPerTierTotal[i]);
}

return tokensMinted;
```

## Get Tokens Sold For

**Summary:** This function allows you to check the total number of tokens that have been sold to investors for ETH.

```
/**


* @notice Return the total no. of tokens sold for ETH
* @return uint256 Total number of tokens sold for ETH


*/
function getTokensSoldFor(FundRaiseType _fundRaiseType) public
view returns (uint256) {
uint256 tokensSold;
for (uint8 i = 0; i < mintedPerTier[uint8(_fundRaiseType)].len
gth; i++) {
tokensSold = tokensSold.add(mintedPerTier[uint8(_fundRaiseTyp
e)][i]);
}
return tokensSold;
```

## Get Number of Tiers

**Summary:** This function allows you to check the total number of tiers that you have in your STO.

```
/**


* @notice Return the total no. of tiers
* @return uint256 Total number of tiers


*/


function getNumberOfTiers() public view returns (uint256) {
return tokensPerTierTotal.length;
}
```

## Get Permissions

**Summary:** This function allows you to check to see if there are any permissions flags raised  in your STO.

```
/**
* @notice Return the permissions flag that are associated with
STO
*/
function getPermissions() public view returns(bytes32[]) {
bytes32[] memory allPermissions = new bytes32[](0);
return allPermissions;
}
```

## Get Init Function

**Summary:** This function shows you the signature of the configure function.

```
/**
```

```
* @notice This function returns the signature of configure function
* @return bytes4 Configure function signature
*/
function getInitFunction() public pure returns (bytes4) {
return 0xb0ff041e;
}
```

## Get Oracle

**Summary:** This function returns the addresses registered and used with the Polymath Registry and the oracle.

```

function _getOracle(bytes32 _currency, bytes32 _denominatedCurrency) internal view returns (address) {


return PolymathRegistry(RegistryUpdater(securityToken).polymathRegistry()).getAddress(oracleKeys[_currency][_denominatedCurrency]);
}
```

## Special considerations / notes

None

## Troubleshooting / FAQs

None

## Know Issues / bugs

None

## Changelog

**2.0.1 (November / December 2018 - Not release yet):**

- **Added** `getSTODetails` to USDTSTO:
  - This function returns all of the details for issuers to review about their USDTieredSTO.
- **Added** an Array of Tiers that will hold data about every tier in USDTSTO.
  - The array of tiers provides all the data for the each tier in their STO for issuer to review and monitor.
- **Added** `buyWithETHRateLimited`, `buyWithPOLYRateLimited` and `buyWithUSDRateLimited` to USDTSTO.
  - These functions create a rate limit when an investor investing in an STO and buying with ETH, POLY or USD (USD pegged stable coin).
- **Added** `getTokensSoldByTier` to return sold (not minted during finalization) tokens in each tier to USDTSTO.
  - This function returns the information regarding the number of tokens sold for each tier of in an STO.
- **Added:** Allow USDTieredSTO to accept multiple stablecoin addresses instead of just a single stable coin address

**2.1.0**

- **Added/Fixed:** With the BuywithUSD function the user needs to provide the stable coin address they want to use when investing.
- **Added/Fixed:** With the Configure function- when the issuer is configuring their STO, they need to provide an array of addresses of stable coins they will accept (changed from earlier as the only option was the DAI stable coin).