Jeff Senk Follow
Smarter Contracts and Decision Engines
May 18 · 5 min read

A General Model for Tokenized Funding and Governance

Blockchain tokens became a wildly popular (and controversial) funding mechanism in 2017, allowing development teams to raise millions of dollars seemingly overnight with little more than a product roadmap. Decentralized platforms like EOS, Augur and Status accepted payments of Bitcoin and Ether in exchange for tokens that could be used on the platform once it was built.

Unfortunately, unscrupulous or inexperienced development teams soon began to abuse this new fundraising channel to launch projects that ultimately went nowhere, leaving investors holding worthless tokens. If the industry is to recapture investor trust (and avoid onerous government regulation), development teams need to adopt better controls and oversight.

We present an alternative to the traditional ICO fundraising model which we call the Equity Token. The Equity Token contract enumerates investor rights such as assigning periodic budgets, distributing dividends, electing an administrator, and choosing to wind down the company if the team fails to reach its goals. The design captures many of the benefits of a traditional corporate structure, while eliminating the legal and bureaucratic overhead that this normally entails. Teams that elect to fundraise through a mechanism with built-in oversight and investor protections send a powerful signal of reputability to the market, which should enable them to raise more money.

Solidity code for a prototype of the Equity Token smart contract can be found here: Github

The Equity Token Contract

Constructor

```
constructor(uint256 initSupply, uint256 _quorum, uint256
_dividendPeriod,    uint256 _budgetPeriod) public{
  totalSupply_ = initSupply;
  quorum = _quorum;
  administrator = msg.sender;
  tokenBalances[administrator] = totalSupply_;
  members.push(administrator);
  /*to initialize elections must put sole ownership support
behind defaults**/
  candidateSupport[administrator] = administrator;
  candidateVotes[administrator] =
tokenBalances[administrator];
  dividendSupport[administrator] = 0;
  dividendVotes[0] = tokenBalances[administrator];
  budgetSupport[administrator] = 0;
  budgetVotes[0] = tokenBalances[administrator];
  lastDividend = now;
  lastBudget = now;
  dividendPeriod = _dividendPeriod;
  budgetPeriod = _budgetPeriod;
}
```

The constructor creates an organization that is entirely owned and controlled by the address that created the contract. As investors send ETH into the contract it is up to the administrator to transfer the correct number of tokens to their address. The constructor also sets the quorum variable, which is the minimum number of token votes required for a motion to pass. This number should be set to a value greater than 50% of the total supply to avoid conflicting decisions. In most cases it would be set to 51%, unless the company desires a higher threshold of consensus.

Token Transfer

```
function transfer(address _to, uint256 _value) public
returns (bool){
  require(_to != address(0));
  require(_value <= tokenBalances[msg.sender]);
  tokenBalances[msg.sender] =
sub(tokenBalances[msg.sender],_value);
  tokenBalances[_to] = add(tokenBalances[_to],_value);
  bool isMember = false;
  for(uint i=0;i<members.length;i++){
    if(members[i] == _to){
      isMember = true;
      break;
    }
  }
  if(!isMember){
    members.push(_to);
  }
  /**May also want to remove msg.sender if
tokenBalances[msg.sender]
  <=0, else history is retained of all previous members*/
  /**Preferences of msg.sender automatically transfer to
_to**/
  dividendSupport[_to] = dividendSupport[msg.sender];
  budgetSupport[_to] = budgetSupport[msg.sender];
  candidateSupport[_to] = candidateSupport[msg.sender];
  emit Transfer(msg.sender, _to, _value);
  return true;
}
```

When tokens are transferred from one owner to another, the function checks if the new owner is already a listed member, and if not adds them to the array. The member list is used to calculate and distribute dividends at the end of a period. The voting preferences of the prior token owner (dividend, budget, and administrator candidate) automatically transfer to the new owner, and must be manually changed if the new owner so chooses. The contract also implements the totalSupply() and balanceOf() functions from the ERC20 specification.

Dividend Functions

```
function electDividend(uint256 _dividend) public returns
(bool){
  uint256 currentSupport = dividendSupport[msg.sender];

dividendVotes[currentSupport]=sub(dividendVotes[currentSupp
ort],tokenBalances[msg.sender]);
  dividendSupport[msg.sender] = _dividend;
  dividendVotes[_dividend] =
  add(dividendVotes[_dividend],tokenBalances[msg.sender]);
  if(dividendVotes[_dividend]>quorum){
    dividend = dividendVotes[_dividend];
  }
  return true;
}
```

The members must collectively vote on the amount of ETH to distribute as dividends at the end of every period (i.e. quarterly or annually), with the remainder held as retained earnings. Members vote by assigning "votes" equal to their total token holdings in support of a specific dividend value. A value receiving more than the quorum of votes becomes the new dividend.

```
  function distributeDividend() public onlyAdmin returns
  (bool){
    require(dividend < address(this).balance);
    require(now > add(lastDividend,dividendPeriod));
    lastDividend = now;
    for(uint i =0;i<members.length;i++){
      uint256 share =
  calcShare(tokenBalances[members[i]],dividend);
      ethBalances[members[i]] =
  add(ethBalances[members[i]],share);
    }
    emit DividendDistributed(dividend);
    dividend = 0;
    return true;
  }
```

If a dividend has been set and sufficient time has passed since the last disbursement, the Administrator may call the distributeDividend() function. ETH is allocated to the member addresses based on the percentage of the total supply they hold. Once a disbursement occurs the function is locked until the next period elapses.

Budget Functions

```
  function electBudget(uint256 _budget) public returns (bool)
  {
    uint256 currentSupport = budgetSupport[msg.sender];
    budgetVotes[currentSupport] =
  sub(budgetVotes[currentSupport],tokenBalances[msg.sender]);
    budgetSupport[msg.sender] = _budget;
    budgetVotes[_budget] =
    add(budgetVotes[_budget],tokenBalances[msg.sender]);
    if(budgetVotes[_budget] > quorum){
      budget = _budget;
    }
    return true;
  }
```

```
function distributeBudget(address _to) public onlyAdmin
returns (bool){
  require(budget <= address(this).balance);
  require(now > add(lastBudget,budgetPeriod));
  lastBudget = now;
  ethBalances[_to] = add(ethBalances[_to],budget);
  emit BudgetDistributed(_to,budget);
  budget = 0;
  return true;
}
```

The system for selecting a periodical budget uses the same voting mechanism as selecting a dividend. Once the amount has been agreed upon, the Administrator calls distributeBudget() and assigns the funds to an external address which will be used for day to day operations of the company. We recommend this be a smart contract address which implements rules for how the funds may be used, for instance a parameter which releases a set amount of ETH per week. Vitalik outlines such a time-based tap system in the DAICO explanation.

Administrator Election

```
function electAdmin(address candidate) public returns
(bool){
  address currentSupport = candidateSupport[msg.sender];
  candidateVotes[currentSupport] =

sub(candidateVotes[currentSupport],tokenBalances[msg.sender
]);
  candidateSupport[msg.sender] = candidate;
  candidateVotes[candidate] =
  add(candidateVotes[candidate],tokenBalances[msg.sender]);
  if(candidateVotes[candidate] > quorum){
    administrator = candidate;
    emit AdminElected(administrator,
candidateVotes[candidate]);
  }
  return true;
}
```

The Administrator is elected by members casting their support for a particular candidate's address. The candidate who receives more than the quorum of votes immediately enjoys Administrative rights. The election may take place at any time by members shifting their support. The Administrator is tasked with disbursing dividends and budget funds, and is akin to the Chairman of the Board.

Self Destruct Mechanism

```
function windDown() public returns (bool){
  windDownVotes =
add(windDownVotes,tokenBalances[msg.sender]);
  if(windDownVotes > quorum){
    for(uint i =0;i<members.length;i++){
      uint256 share = address(this).balance
      *(tokenBalances[members[i]]/totalSupply_);
      /**rounding*/
      ethBalances[members[i]] =
add(ethBalances[members[i]],share);
    }
  }
  return true;
}
```

If at any time the members become dissatisfied with the performance of the team, they may elect to wind down the company. A passing vote to wind down immediately assigns all of the remaining ETH to the member addresses in proportion to their holdings. The contract itself is not destroyed (in part so the members have time to withdraw funds), but there is no further budget for the team to work with.

Potential Pitfalls and Improvements

Looping is generally a bad idea in Ethereum contracts, and if the list of members grows too large there is a chance that the functions which iterate over the array could run out of gas. It may also be useful to add different quorum thresholds for different types of vote, and a mechanism for updating quorum values and voting periods.

This contract is a prototype and we strongly recommend against committing real funds to a project of this design until it has been audited. If you come across any bugs or would like to suggest a feature please leave a comment on this post or open an issue on Github. We very much welcome feedback as we work to create more open, transparent, and equitable working relationships.

Many thanks to Eric Olszewski for contributing his thoughts to this piece.