February 19

Daly Ventures

Blockchain Powered Investment Fund Whitepaper (1.0)

109 St Marks Place, New York, NY10009

Table of Contents

Fund Structure

- Purpose of Fund 1 is to create long term generational wealth for token holders and portfolio company employees.
- In order to enable the movement of capital between each entity:
    - Investors can only invest at fund level
    - Portfolio company leaders will own tokens at the Fund Level. This way their equity value is maximized when the Fund is optimal vs. when their individual entity is performing
- Company shareholders will have to decide for themselves whether these investment terms and operating principles pose a risk to long-term performance or whether they are a source of competitive advantage that will be sustained for years in the future.

(1)Fund 1:

- Focused on seed stage incubation and investment.
- Investors will have first right to invest in all follow-on rounds (where possible)
- Potential Investments opportunities currently: Irish Whiskey Distillery and Real Estate Time Sharing for Consulting Firms / Select Corporations.

Ownership Principles and Policies

1. Daly Ventures will be a diverse portfolio whose long-term goal is to maximize its annual rate of gain in intrinsic business value on a per share basis. Daly Ventures preference will own a diversified group of businesses that generate profit and consistently generate various distributions to all token holders
2. All business will be conducted through separately incorporated subsidiaries whose Founders will operate with extreme autonomy.
3. Daly Ventures Management Team assume the following responsibilities;
    a) Facilitate the voting process to make all security investments
    b) Elect/Approve Leadership of important subsidiaries, fixing compensation and obtain from each a private recommendation for a successor in case one was suddenly needed.
    c) Deploy most cash not needed in subsidiaries after they had increased their competitive advantage, with the ideal deployment being the use of that cash to acquire new subsidiaries.
4. Operate with a model based on extreme decentralization of operating authority, with responsibility for business performance placed entirely in the hands of local managers and investment decisions shared across all limited partners (token holders).

5. As an important matter of preferred conduct, Daly Ventures will almost never spin off a portfolio company. Voluntary spin-offs, make no sense in most situations as we would lose control, capital-allocation flexibility and potential tax advantages. Moreover, the parent and the spun-off operations, once separated, would likely incur moderately greater costs than existed when they were combined.
6. Daly Ventures will have little debt outstanding as it maintains (i) creditworthiness under all conditions and (ii) easy availability of cash and credit for deployment in times presenting unusual opportunities.

Advantages of proposed Structure:

- Without incurring taxes or much in the way of other costs, provides structure to move huge sums of value from businesses that have limited opportunities for incremental investment to other sectors with greater promise.
- Our portfolio companies are more valuable as part of Daly Ventures than as separate entities. One reason is our ability to move funds between businesses or into new ventures instantly and without tax. In addition, certain costs duplicate themselves, in full or part, if operations are separated.

Corporate Governance

Daly Ventures will have two requirements for operating managers:

1. Submit financial statement information on a monthly basis and
2. Send free cash flow generated by operations to headquarters

Portfolio managers can speak to the Daly Ventures Management Team as frequently or infrequently as they like, but they alone are responsible for all operating decisions. They will focus entirely on improving the long-term economics of their business, and all decisions that are made from the standpoint of their impact on competitive positioning, long term profitability and free cash flow.

Voting Process

- One vote will be received for every token owned
- Supermajority required for any investment
- All Token Holders will vote to approve any fund
- 65%. or above approves the vote

Compensation for Portfolio Managers

Managers at Daly Venture portfolio companies will be paid modest salaries but stand to receive very significant bonuses if annual performance goals are achieved. The formula for determining this payout varies by business and is tailored so that it was appropriate for that company's complexity and capital requirements. Compensation bonuses will place heavy emphasis on manager's success in generating and delivering profit to the fund for relocation. The bonus formula will not be capped.

Compensation Formula:
20% EBITDA distributed to the DayBlink Investment fund

Stock Purchase Option:
(**Δ** Company Valuation / **Δ** Portfolio Valuation) will determine purchase amount

Deal Investment Process

(1)The Deal arrives

- Assigned to Partner based on experience in the sector/contact with the CEO/Deal Source
- This person has the option to turn down the deal immediately or investigate it.
- Partner does some preliminary research (reference checks etc.) and then if interested, looks to recruit a second, which might be a partner or principle.
    - If the partner cannot convince one other person in the firm to support the deal the deal does not make it to the next step

(2)The Deal becomes active

- Partner writes a memo from 1-5 pages introducing the company to the firm and profiling the company's market, team and technology.
- The sponsor of the deal team will solicit questions and feedback from the management team and work to resolve them. 3-4 key risks will be identified and mitigated before an official recommendation is made.

(3)The Deal generates positive sentiment

- Partners are aware of the investment opportunity and supported an investment subject to certain terms or outstanding questions.
- Depending on the response to key questions the deal will be put forward to consideration or abandoned.

(4) An investment is recommended

- Before an investment decision is made to fund a company, an investment memo will be created and shared with all token holders outlining:
    - Technical Assessment
    - Competitive Analysis
    - Market Opportunity
    - Projected Financials
- In all cases the final decision will be made by vote - based on weighted votes.

(5) A vote is scheduled

- Company presents opportunity to all token holders. Voting regulations are applied.

APPENDIX

Daly Ventures Token Code:

```
pragma
solidity
>=0.4.22
<0.6.0;

        contract owned {
            address public owner;

            constructor() public {
                owner = msg.sender;
            }

            modifier onlyOwner {
                require(msg.sender == owner);
                _;
            }

            function transferOwnership(address newOwner) onlyOwner public {
                owner = newOwner;
            }
        }

        interface tokenRecipient { function receiveApproval(address _from, uint256 _value, address _token,
        bytes _extraData) external;

        }
        contract Dalycoin {
            // Public variables of the token
            string public name;
            string public symbol;
            uint8 public decimals = 18;
            // 18 decimals is the strongly suggested default, avoid changing it
            uint256 public totalSupply;

            // This creates an array with all balances
            mapping (address => uint256) public balanceOf;
            mapping (address => mapping (address => uint256)) public allowance;
            mapping (address => bool) public frozenAccount;

            // This generates a public event on the blockchain that will notify clients
```

```
event Transfer(address indexed from, address indexed to, uint256 value);

// This generates a public event on the blockchain that will notify clients
event Approval(address indexed _owner, address indexed _spender, uint256 _value);

// This notifies clients about the amount burnt
event Burn(address indexed from, uint256 value);

/**
 * Constructor function
 *
 * Initializes contract with initial supply tokens to the creator of the contract
 */
constructor(

) public {
    totalSupply = 22000000 * 10 ** uint256(18);  // Update total supply with the decimal amount
    balanceOf[msg.sender] = totalSupply;                // Give the creator all initial tokens
    name = "Dalycoin";                                  // Set the name for display purposes
    symbol = "DLY";                             // Set the symbol for display purposes
}

/**
 * Internal transfer, only can be called by this contract
 */
function _transfer(address _from, address _to, uint _value) internal {
    // Prevent transfer to 0x0 address. Use burn() instead
    require(_to != address(0x0));
    // Check if the sender has enough
    require(balanceOf[_from] >= _value);
    // Check for overflows
    require(balanceOf[_to] + _value > balanceOf[_to]);
    // Save this for an assertion in the future
    uint previousBalances = balanceOf[_from] + balanceOf[_to];
    // Subtract from the sender
    balanceOf[_from] -= _value;
    // Add the same to the recipient
    balanceOf[_to] += _value;
    emit Transfer(_from, _to, _value);
    // Asserts are used to use static analysis to find bugs in your code. They should never fail
    assert(balanceOf[_from] + balanceOf[_to] == previousBalances);
}
```

```
/**
 * Transfer tokens
 *
 * Send `_value` tokens to `_to` from your account
 *
 * @param _to The address of the recipient
 * @param _value the amount to send
 */
function transfer(address _to, uint256 _value) public returns (bool success) {
    _transfer(msg.sender, _to, _value);
    return true;
}

/**
 * Transfer tokens from other address
 *
 * Send `_value` tokens to `_to` in behalf of `_from`
 *
 * @param _from The address of the sender
 * @param _to The address of the recipient
 * @param _value the amount to send
 */
function transferFrom(address _from, address _to, uint256 _value) public returns (bool success) {
    require(_value <= allowance[_from][msg.sender]);     // Check allowance
    allowance[_from][msg.sender] -= _value;
    _transfer(_from, _to, _value);
    return true;
}

/**
 * Set allowance for other address
 *
 * Allows `_spender` to spend no more than `_value` tokens in your behalf
 *
 * @param _spender The address authorized to spend
 * @param _value the max amount they can spend
 */
function approve(address _spender, uint256 _value) public
    returns (bool success) {
    allowance[msg.sender][_spender] = _value;
    emit Approval(msg.sender, _spender, _value);
    return true;
```

```
    }

    /**
     * Set allowance for other address and notify
     *
     * Allows `_spender` to spend no more than `_value` tokens in your behalf, and then ping the contract
about it
     *
     * @param _spender The address authorized to spend
     * @param _value the max amount they can spend
     * @param _extraData some extra information to send to the approved contract
     */
    function approveAndCall(address _spender, uint256 _value, bytes memory _extraData)
        public
        returns (bool success) {
        tokenRecipient spender = tokenRecipient(_spender);
        if (approve(_spender, _value)) {
            spender.receiveApproval(msg.sender, _value, address(this), _extraData);
            return true;
        }
    }

    /**
     * Destroy tokens
     *
     * Remove `_value` tokens from the system irreversibly
     *
     * @param _value the amount of money to burn
     */
    function burn(uint256 _value) public returns (bool success) {
        require(balanceOf[msg.sender] >= _value);   // Check if the sender has enough
        balanceOf[msg.sender] -= _value;            // Subtract from the sender
        totalSupply -= _value;                      // Updates totalSupply
        emit Burn(msg.sender, _value);
        return true;
    }
    /**
     * Destroy tokens from other account
     *
     * Remove `_value` tokens from the system irreversibly on behalf of `_from`.
     *
     * @param _from the address of the sender
     * @param _value the amount of money to burn
```

```
     */
    function burnFrom(address _from, uint256 _value) public returns (bool success) {
        require(balanceOf[_from] >= _value);                // Check if the targeted balance is enough
        require(_value <= allowance[_from][msg.sender]);    // Check allowance
        balanceOf[_from] -= _value;                         // Subtract from the targeted balance
        allowance[_from][msg.sender] -= _value;             // Subtract from the sender's allowance
        totalSupply -= _value;                              // Update totalSupply
        emit Burn(_from, _value);
        return true;
    }
}
```

Voting Code:

```
pragma solidity >=0.4.22 <0.6.0;

contract owned {
    address public owner;

    constructor() public {
        owner = msg.sender;
    }

    modifier onlyOwner {
        require(msg.sender == owner);
        _;
    }

    function transferOwnership(address newOwner) onlyOwner  public {
        owner = newOwner;
    }
}

contract tokenRecipient {
    event receivedEther(address sender, uint amount);
    event receivedTokens(address _from, uint256 _value, address _token, bytes _extraData);

    function receiveApproval(address _from, uint256 _value, address _token, bytes memory _extraData) public {
        Token t = Token(_token);
        require(t.transferFrom(_from, address(this), _value));
        emit receivedTokens(_from, _value, _token, _extraData);
    }

    function () payable external {
        emit receivedEther(msg.sender, msg.value);
    }
}

interface Token {
    function transferFrom(address _from, address _to, uint256 _value) external returns (bool success);
}

contract Congress is owned, tokenRecipient {
    // Contract Variables and events
    uint public minimumQuorum;
    uint public debatingPeriodInMinutes;
    int public majorityMargin;
    Proposal[] public proposals;
    uint public numProposals;
    mapping (address => uint) public memberId;
    Member[] public members;

    event ProposalAdded(uint proposalID, address recipient, uint amount, string description);
    event Voted(uint proposalID, bool position, address voter, string justification);
    event ProposalTallied(uint proposalID, int result, uint quorum, bool active);
    event MembershipChanged(address member, bool isMember);
    event ChangeOfRules(uint newMinimumQuorum, uint newDebatingPeriodInMinutes, int newMajorityMargin);

    struct Proposal {
        address recipient;
        uint amount;
        string description;
        uint minExecutionDate;
        bool executed;
        bool proposalPassed;
        uint numberOfVotes;
```

```
    int currentResult;
    bytes32 proposalHash;
    Vote[] votes;
    mapping (address => bool) voted;
}

struct Member {
    address member;
    string name;
    uint memberSince;
}

struct Vote {
    bool inSupport;
    address voter;
    string justification;
}

// Modifier that allows only shareholders to vote and create new proposals
modifier onlyMembers {
    require(memberId[msg.sender] != 0);
    _;
}

/**
 * Constructor
 */
constructor (
    uint minimumQuorumForProposals,
    uint minutesForDebate,
    int marginOfVotesForMajority
)  payable public {
    changeVotingRules(minimumQuorumForProposals, minutesForDebate, marginOfVotesForMajority);
    // It's necessary to add an empty first member
    addMember(address(0), "");
    // and let's add the founder, to save a step later
    addMember(owner, 'founder');
}

/**
 * Add member
 *
 * Make `targetMember` a member named `memberName`
 *
 * @param targetMember ethereum address to be added
 * @param memberName public name for that member
 */
function addMember(address targetMember, string memory memberName) onlyOwner public {
    uint id = memberId[targetMember];
    if (id == 0) {
        memberId[targetMember] = members.length;
        id = members.length++;
    }

    members[id] = Member({member: targetMember, memberSince: now, name: memberName});
    emit MembershipChanged(targetMember, true);
}

/**
 * Remove member
 *
 * @notice Remove membership from `targetMember`
 *
 * @param targetMember ethereum address to be removed
 */
```

```
    function removeMember(address targetMember) onlyOwner public {
        require(memberId[targetMember] != 0);

        for (uint i = memberId[targetMember]; i<members.length-1; i++){
            members[i] = members[i+1];
            memberId[members[i].member] = i;
        }
        memberId[targetMember] = 0;
        delete members[members.length-1];
        members.length--;
    }

    /**
     * Change voting rules
     *
     * Make so that proposals need to be discussed for at least `minutesForDebate/60` hours,
     * have at least `minimumQuorumForProposals` votes, and have 50% + `marginOfVotesForMajority` votes to be
executed
     *
     * @param minimumQuorumForProposals how many members must vote on a proposal for it to be executed
     * @param minutesForDebate the minimum amount of delay between when a proposal is made and when it can be
executed
     * @param marginOfVotesForMajority the proposal needs to have 50% plus this number
     */
    function changeVotingRules(
        uint minimumQuorumForProposals,
        uint minutesForDebate,
        int marginOfVotesForMajority
    ) onlyOwner public {
        minimumQuorum = minimumQuorumForProposals;
        debatingPeriodInMinutes = minutesForDebate;
        majorityMargin = marginOfVotesForMajority;

        emit ChangeOfRules(minimumQuorum, debatingPeriodInMinutes, majorityMargin);
    }

    /**
     * Add Proposal
     *
     * Propose to send `weiAmount / 1e18` ether to `beneficiary` for `jobDescription`. `transactionBytecode ? Contains :
Does not contain` code.
     *
     * @param beneficiary who to send the ether to
     * @param weiAmount amount of ether to send, in wei
     * @param jobDescription Description of job
     * @param transactionBytecode bytecode of transaction
     */
    function newProposal(
        address beneficiary,
        uint weiAmount,
        string memory jobDescription,
        bytes memory transactionBytecode
    )
        onlyMembers public
        returns (uint proposalID)
    {
        proposalID = proposals.length++;
        Proposal storage p = proposals[proposalID];
        p.recipient = beneficiary;
        p.amount = weiAmount;
        p.description = jobDescription;
        p.proposalHash = keccak256(abi.encodePacked(beneficiary, weiAmount, transactionBytecode));
        p.minExecutionDate = now + debatingPeriodInMinutes * 1 minutes;
        p.executed = false;
        p.proposalPassed = false;
        p.numberOfVotes = 0;
```

```
        emit ProposalAdded(proposalID, beneficiary, weiAmount, jobDescription);
        numProposals = proposalID+1;

        return proposalID;
    }

    /**
     * Add proposal in Ether
     *
     * Propose to send `etherAmount` ether to `beneficiary` for `jobDescription`. `transactionBytecode ? Contains : Does
not contain` code.
     * This is a convenience function to use if the amount to be given is in round number of ether units.
     *
     * @param beneficiary who to send the ether to
     * @param etherAmount amount of ether to send
     * @param jobDescription Description of job
     * @param transactionBytecode bytecode of transaction
     */
    function newProposalInEther(
        address beneficiary,
        uint etherAmount,
        string memory jobDescription,
        bytes memory transactionBytecode
    )
        onlyMembers public
        returns (uint proposalID)
    {
        return newProposal(beneficiary, etherAmount * 1 ether, jobDescription, transactionBytecode);
    }

    /**
     * Check if a proposal code matches
     *
     * @param proposalNumber ID number of the proposal to query
     * @param beneficiary who to send the ether to
     * @param weiAmount amount of ether to send
     * @param transactionBytecode bytecode of transaction
     */
    function checkProposalCode(
        uint proposalNumber,
        address beneficiary,
        uint weiAmount,
        bytes memory transactionBytecode
    )
        view public
        returns (bool codeChecksOut)
    {
        Proposal storage p = proposals[proposalNumber];
        return p.proposalHash == keccak256(abi.encodePacked(beneficiary, weiAmount, transactionBytecode));
    }

    /**
     * Log a vote for a proposal
     *
     * Vote `supportsProposal? in support of : against` proposal #`proposalNumber`
     *
     * @param proposalNumber number of proposal
     * @param supportsProposal either in favor or against it
     * @param justificationText optional justification text
     */
    function vote(
        uint proposalNumber,
        bool supportsProposal,
        string memory justificationText
    )
```

```
        onlyMembers public
        returns (uint voteID)
    {
        Proposal storage p = proposals[proposalNumber]; // Get the proposal
        require(!p.voted[msg.sender]);                // If has already voted, cancel
        p.voted[msg.sender] = true;                   // Set this voter as having voted
        p.numberOfVotes++;                            // Increase the number of votes
        if (supportsProposal) {                       // If they support the proposal
            p.currentResult++;                        // Increase score
        } else {                                      // If they don't
            p.currentResult--;                        // Decrease the score
        }

        // Create a log of this event
        emit Voted(proposalNumber,  supportsProposal, msg.sender, justificationText);
        return p.numberOfVotes;
    }

    /**
     * Finish vote
     *
     * Count the votes proposal #`proposalNumber` and execute it if approved
     *
     * @param proposalNumber proposal number
     * @param transactionBytecode optional: if the transaction contained a bytecode, you need to send it
     */
    function executeProposal(uint proposalNumber, bytes memory transactionBytecode) public {
        Proposal storage p = proposals[proposalNumber];

        require(now > p.minExecutionDate                                // If it is past the voting deadline
            && !p.executed                                          // and it has not already been executed
            && p.proposalHash == keccak256(abi.encodePacked(p.recipient, p.amount, transactionBytecode))  // and the
supplied code matches the proposal
            && p.numberOfVotes >= minimumQuorum);                       // and a minimum quorum has been reached...

        // ...then execute result

        if (p.currentResult > majorityMargin) {
            // Proposal passed; execute the transaction

            p.executed = true; // Avoid recursive calling

            (bool success, ) = p.recipient.call.value(p.amount)(transactionBytecode);
            require(success);

            p.proposalPassed = true;
        } else {
            // Proposal failed
            p.proposalPassed = false;
        }

        // Fire Events
        emit ProposalTallied(proposalNumber, p.currentResult, p.numberOfVotes, p.proposalPassed);
    }
}
```