Ethereum Alarm Clock Documentation

Release 1.0.0

Piper Merriam

October 08, 2015

Contents

The Ethereum Alarm Clock is a service that allows scheduling of contract function calls at a specified block number in the future. These scheduled calls are then executed by other nodes on the ethereum network who are reimbursed for their gas costs plus a small payment for the transaction.

Contents:

CHAPTER 1

Overview

The Ethereum Alarm service is a contract on the ethereum network that facilitates scheduling of function calls for a specified block in the future. It is designed to require little or no trust between any of the users of the service, as well as providing no special access to the creator of the contract.

1.1 Scheduling Function Calls

When a contract, or individual wants to schedule a function call with the Alarm service it will perform the following steps.

1. Ensure that the account that is scheduling the call has a sufficient balance to pay for the scheduled call.
2. Register any call data that will be required to make the function call.
3. Schedule the function call with the service.

1.1.1 Account Balance

The Alarm service operates under a *scheduler pays* model meaning that the address which schedules the function call is required to pay for it. When a call is executed, the initial gas cost is paid for by the ethereum address that sends the executing transaction. This address needs to be reimbursed this gas cost plus a small fee. The Alarm service requires this payment up front in the form of an account balance.

The Alarm service maintains accounts for each address on the network. These accounts can have ether deposited and withdrawn at any time. However, at the time the call is executed, if the scheduler's account does not have enough funds to pay for the execution of the scheduled call, it will be skipped.

1.1.2 Registering Call Data

The Alarm service is not aware of the function ABI for the calls it executes. Instead, it uses the function ABI signature and raw call data to execute the function call.

To do this, any data that needs to be used in the call must be registered prior to scheduling the call. Call data only needs to be registered once, and can be re-used for subsequent function calls.

1.1.3 Call Scheduling

Function calls can be scheduled for any block at least 40 blocks *(~10 minutes)* in the future. Scheduling is done by providing the Alarm service with the following information:

1. Contract address the call should be executed on.
2. ABI signature of the function that should be called.
3. SHA3 hash of the call data that should be included in the function call.
4. Target block number that the call should be executed on.
5. Number of blocks after the target block during which it still ok to execute the call.
6. A nonce to allow differentiation between identical calls that are scheduled for the same block.

Once scheduled, the call waits to be picked up and executed at the desired block.

1.2 Execution of scheduled calls

Scheduled function calls can ultimately be executed by anyone who wishes to initiate the transaction. This will likely be an automated process that monitors for upcoming scheduled calls and executes them at the appropriate block.

1.2.1 Usage Fees

A scheduled function call costs approximately 102% of the total gas expenditure for the transaction in which it was executed.

The additional 2% is split evenly between paying the account which executed the function call and the creator of the Alarm service for the many many hours spent creating it.

1.3 Guarantees

1.3.1 Will the call happen?

There are no guarantees that your function will be called. The design of this service is meant to provide the proper motivation for calls to be executed, but it is entirely possible that certain calls will be missed due to unforseen circumstances.

1.3.2 Will I get paid for executing a call?

If you are diligent about how you go about executing scheduled calls, there should be a near 0% chance that you will not be reimbursed for your gas costs. See the section on executing calls for more information on how to protect yourself.

CHAPTER 2

Account Managment

The *scheduler pays* system requires that payment for scheduled calls be provided prior to the execution of the call, so that the sender of the executing transaction can immediately be reimbursed for the gas costs.

The account and associated funds are used to pay for any calls scheduled by that address. Inturn, each ethereum address may withdraw or deposit funds in its account at any time with no restrictions.

It is also possible to deposit funds in the account of another address. You cannot however withdraw funds from any address other than your own.

2.1 Checking account balance

Your account balance can be checked by accessing the public mapping of accounts to balances.

- **Solidity Function Signature:** `accountBalances(address accountAddress) returns (uint)`
- **ABI Signature:** `0x6ff96d17`

Calling this function will return the balance in wei for the provided address.

2.2 Depositing funds

Depositing funds can be done one of a few ways.

2.2.1 By sending ether

The simplest way to add funds to your account is to just send the ether to the address of the alarm service. Any funds sent to the alarm service are added to the account balance of the sender.

Warning: Contracts cannot add funds to their accounts this way using the `send` function on addresses. This is due to solidity's protection against unbounded gas use in contract fallback functions. See below for how contracts can add their own funds.

Here is how you would do this from the geth javascript console.

The above would deposit 100 wei in the account of whatever address you used for the `from` value in the transaction.

2.2.2 By using the deposit function

Funds can also be deposited in a specific account by calling the `deposit` function and sending the desired deposit value with the transaction.

- **Solidity Function Signature:** `deposit(address accountAddress)`
- **ABI Signature:** `0xf340fa01`

2.2.3 Sending from a contract

Contracts can deposit funds through these mechanisms as well.

Or, if you would like your contract to deposit funds in the account of another address.

Note: It should be pointed out that you cannot deposit funds by calling `alarmAddress.send(value)`. By default in solidity, this transaction is sent with only enough gas to execute the funds transfer, and the fallback function on the Alarm service requires a bit more gas so that it can record the increase in account balance.

2.3 Withdrawing funds

Withdrawing funds is restricted to the address they are associated with. This is done by calling the `withdraw` function on the Alarm service.

- **Solidity Function Signature:** `withdraw(uint value)`
- **ABI Signature:** `2e1a7d4d`

If the account has a balance sufficient to fulfill the request, the amount specified specified in wei will be transferred to `msg.sender`.

CHAPTER 3

Authorization

Scheduled calls can be considered either **authorized** or **unauthorized**.

An **authorized** call is one for which either `scheduledBy == targetAddress` or for which the `scheduledBy` address has been granted explicit authorization by `targetAddress` to schedule calls.

An **unauthorized** call is one for which `scheduledBy != targetAddress` and the `scheduledBy` address has not been granted authorization to schedule calls.

Note: Any address can still schedule calls towards any other address. The authorization status only effects which address the calls originate from, not whether they will be executed.

3.1 Differentiating calls

When the Alarm service executes calls, they will come from one of two addresses depending on whether the call is considered **authorized** or **unauthorized**. These addresses will sometimes be referred to as relays, as they relay the actually function call for the Alarm service, allowing the callee to differentiate between **authorized** and **unauthorized** calls.

Note: A call's authorization state is determined at the time of execution.

authorized calls will orignate from the address returned by the `authorizedAddress` function.

- **Solidity Function Signature:** `authorizedAddress() returns (address)`
- **ABI Signature:** `0x5539d400`

unauthorized calls will orignate from the address returned by the `unauthorizedAddress` function.

- **Solidity Function Signature:** `unauthorizedAddress() returns (address)`
- **ABI Signature:** `0x94d2b21b`

3.2 Checking authorization status

When a function is called on a contract, it can check whether or not it is authorized by checking which of the two relay addresses matches `msg.sender`.

To do this, our contract needs to be at least partially aware of the Alarm ABI function signatures which can be done easily with an abstract contract.

Consider the idea of a contract which holds onto funds until a specified future block at which point it suicides sending all of the funds to the trustee.

```
contract AlarmAPI {
    function authorizedAddress() returns (address);
    function unauthorizedAddress() returns (address);
}

contract TrustFund {
    address trustee = 0x...;
    address _alarm = 0x...;

    function releaseFunds() public {
        AlarmAPI alarm = AlarmAPI(_alarm);
        if (msg.sender == alarm.authorizedAddress()) {
            suicide(trustee);
        }
    }
}
```

In the above example, the `TrustFund.releaseFunds` function checks whether the incoming call is from the **authorized** alarm address before suiciding and releasing the funds.

Note: It should be noted that the above example would require authorization to have been setup by the `TrustFund` contract via some mechanism like a contract constructor.

3.3 Managing Authorization

It is the sole responsibility of the contract to manage address authorizations, as the functions surrounding authorization use `msg.sender` as the `contractAddress` value.

3.3.1 Granting Authorization

Authorization is granted with the `addAuthorization` function.

- **Solidity Function Signature:** `addAuthorization(address schedulerAddress)`
- **ABI Signature:** `0x35b28153`

This function adds the `schedulerAddress` address to the authorized addresses for `msg.sender`.

Here is how a solidity contract could grant access to it's creator.

```
contract Example {
    address alarm = 0x....;

    function Example() {
        alarm.call(bytes4(sha3("addAuthorization(address)")), msg.sender);
    }
}
```

Upon creation, the `Example` contract adds it's creator as an authorized scheduler with the alarm service.

3.3.2 Checking Access

You can check whether an address has authorization to schedule calls for a given address with the `checkAuthorization` function.

- **Solidity Function Signature:** `checkAuthorization(address schedulerAddress, address contractAddress) returns (bool)`
- **ABI Signature:** `0x685c234a`

3.3.3 Removing Authorization

A contract can remove authorization from a given address using the `removeAuthorization` function.

- **Solidity Function Signature:** `removeAuthorization(address schedulerAddress)`
- **ABI Signature:** `0x94f3f81d`

```
contract MemberRoster {
    address alarm = 0x....;

    mapping (address => bool) members;

    function removeMember(address memberAddress) {
        members[memberAddress] = false;

        alarm.call(bytes4(sha3("removeAuthorization(address)")), memberAddress);
    }
}
```

In the example above we are looking at part of a contract that manages the membership for an organization of some sort. Upon removing a member from the organization, the `MemberRoster` contract also removes their authorization status for scheduled calls.

CHAPTER 4

Scheduling

Call scheduling is the core of the Ethereum Alarm Service. Calls can be scheduled on any block at least 40 blocks *(~10 minutes)* in the future.

4.1 Registering Call Data

If a function call requires arguments, then prior to scheduling the call, the call data for those arguments must be registered. This is done with the `registerData` function.

- **Solidity Function Signature:** `registerData()`
- **ABI Signature:** `0xb0f07e44`

It may be confusing at first to see that this function does not take any arguments, yet it is responsible for recording the call data for a future function call. Internally, the `registerData` function pulls the call data off of `msg.data`, effectively allowing any number and type of arguments to be passed to it (like the sha3 function).

In solidity, this would look something like the following.

Upon receiving this call, the Alarm service strips off the first four bytes from `msg.data` to remove the ABI function signature and then stores the full call data.

Call data only ever needs to be registered once after which it can be used without needing to re-register it.

The `registerData` function cannot be used via an abstract contract in solidity, as solidity has not mechanism to allow for variadic arguments in a function call. You can however, simplify some of your contract code with a local alias on your contract that handles the `call` logic for you.

You can implement as many local `registerData` functions as you need with each argument pattern that you need to schedule data for, allowing for simple data registration.

4.2 Scheduling the Call

Function calls are scheduled with the `scheduleCall` function on the Alarm service.

- **Solidity Function Signature:** `scheduleCall(address contractAddress, bytes4 signature, bytes32 dataHash, uint targetBlock, uint8 gracePeriod, uint nonce);`
- **ABI Signature:** `0x52afbc33`

The `scheduleCall` function takes the following parameters:

- **address contractAddress:** The contract address that the function should be called on.
- **bytes4 abiSignature:** The 4 byte ABI function signature for the call.
- **bytes32 dataHash:** The `sha3` hash of the call data for the call.
- **uint targetBlock:** The block number the call should be executed on.
- **uint8 gracePeriod:** The number of blocks after `targetBlock` that it is ok to still execute this call.
- **uint nonce:** Number to allow for differentiating a call from another one which has the exact same information for all other user specified fields.

Note: Prior to scheduling a function call, any call data necessary for the call must have already been registered.

The `scheduleCall` function has two alternate invocation formats that can be used as well.

- **Solidity Function Signature:** `scheduleCall(address contractAddress, bytes4 abiSignature, bytes32 dataHash, uint targetBlock, uint8 gracePeriod) public`
- **ABI Signature:** `0x1145a20f`

When invoked this way, the **nonce** argument is defaulted to `0`.

- **Solidity Function Signature:** `scheduleCall(address contractAddress, bytes4 abiSignature, bytes32 dataHash, uint256 targetBlock) public`
- **ABI Signature:** `0xf828c3fa`

When invoked this way, the **gracePeriod** argument is defaulted to `255` and then **nonce** set to `0`.

4.2.1 Contract scheduling its own call

Contracts can take care of their own call scheduling.

In this example `Lottery` contract, every time the `beginLottery` function is called, a call to the `pickWinner` function is scheduled for approximately 24 hours later (5760 blocks).

4.2.2 Scheduling a call for a contract

Alternatively, calls can be scheduled to be executed on other contracts

---

Note: The Alarm service operates under a *scheduler pays* model meaning that payment for all executed calls is taken from the scheduler's account.

Lets look at an example where we want to schedule a funds transfer for a wallet contract of some sort.

Note: This example assuming that you have the Alarm contract ABI loaded into a web3 contract object.

```
// First register the call data
// 0xb0f07e44 is the ABI signature for the `registerData` function.
> callData = ...  // the full ABI encoded call data for the call we want to schedule.
> web3.sendTransaction({to: alarm.address, data: 'b0f07e44' + callData, from: eth.coinbase})
// Now schedule the call
> dataHash = eth.sha3(callData)
> signature = ... // the 4-byte ABI function signature for the wallet function that transfers funds.
```

```
> targetBlock = eth.getBlock('latest') + 100  // 100 blocks in the future.
> alarm.scheduleCall.sendTransaction(walletAddress, signature, dataHash, targetBlock, 255, 0, {from:
```

There is a lot going on in this example so lets look at it line by line.

1. `callData = ...`

   Our wallet contract will likely take some function arguments when transferring funds, such as the amount to be transferred. This variable would need to be populated with the ABI encoded call data for this function.

2. `web3.sendTransaction({to: alarm.address, data: 'b0f07e44' + callData, from: eth.coinbase})`

   Here we are registering the call data with the Alarm service. `b0f07e44` is the ABI encoded call signature for the `registerData` function on the alarm service.

3. `dataHash = eth.sha3(callData)`

   Here we compute the `sha3` hash of the call data we will want sent with the scheduled call.

4. `signature = ...`

   We also need to tell the Alarm service the 4 byte function signature it should use for the scheduled call. Assuming our wallet's transfer function had a call signature of `transferFunds(address to, uint value)` then this value would be the result of `bytes4(sha3(transferFunds(address,uint256))`.

5. `targetBlock = eth.getBlock('latest') + 100`

   Schedule the call for 100 blocks in the future.

6. `alarm.scheduleCall.sendTransaction(walletAddress, signature, dataHash, targetBlock, 255, 0, {from: eth.coinbase})`

   This is the actual line that schedules the function call. We send a transaction using the `scheduleCall` function on the Alarm contract telling the Alarm service to schedule the call for 100 blocks in the future with the maximum grace period of 255 blocks, and a nonce of 0.

It should be noted that this example does not take into account any of the authorization issues that would likely need to be in place such as restricting the tranfer funds function to only accept authorized calls as well as authorizing the desired addresses to make calls to the wallet address.

4.3 Cancelling a call

A scheduled call can be cancelled by its scheduler up to 4 blocks (2 minutes) before it's target block. To cancel a scheduled call use the `cancelCall` function.

- **Solidity Function Signature:** `cancelCall(bytes32 callKey)`
- **ABI Signature:** `0x60b831e5`

CHAPTER 5

Caller Pool

The Alarm service maintains a pool of bonded callers who are responsible for executing scheduled calls. By joining the caller pool, an account is committing to executing scheduled calls in a reliable and consistent manner. Any caller who reliably executes the calls which are allocated to them will make a consistent profit from doing so, while callers who don't get removed from the pool and forfeit some or all of their bond.

The Caller Pool is handled by a separate contract. This contract is deployed by the Alarm service contract during creation. The address of this contract can be retrieved with the `getCallerPoolAddress` function on the Alarm service.

- **Solidity Function Signature:** `getCallerPoolAddress() returns (address)`
- **ABI Signature:** `0x662fc8a0`

This returns the appropriate address to use for interacting with the Caller Pool.

5.1 Caller Bonding

In order to execute scheduled calls, callers put up a small amount of ether up front. This bond, is held for the duration that a caller remains in the caller pool.

5.1.1 Minimum Bond

The bond amount is set as the maximum allowed transaction cost for a given block. This value can be retrieved with the `getMinimumBond` function.

- **Solidity Function Signature:** `getMinimumBond() returns (uint)`
- **ABI Signature:** `0x23306ed6`

This value can change from block to block depending on the gas price and gas limit.

5.1.2 Depositing your bond

Use the `depositBond` function on the Caller Pool to deposit ether towards your bond.

- **Solidity Function Signature:** `depositBond()`
- **ABI Signature:** `0x741b3c39`

5.1.3 Checking bond balance

Use the `callerBonds` function to check the balance of your bond.

- **Solidity Function Signature:** `callerBonds(address) returns (uint)`
- **ABI Signature:** `0xc861cd66`

5.1.4 Withdrawing your bond

Use the `withdrawBond` function on the Caller Pool to withdraw the bond ether.

- **Solidity Function Signature:** `withdrawBond()`
- **ABI Signature:** `0xc3daab96`

If you are currently in a call pool, either active or queued, you will not be able to withdraw your account balance below the minimum bond amount.

5.2 Bond Forfeiture

In the event that a caller fails to execute a scheduled call during their allocated call window, a portion of their bond is forfeited and they are removed from the caller pool. The amount forfeited is equal to the current minimum bond amount.

There are no restrictions on re-entering the caller pool as long as a caller is willing to put up a new bond.

5.3 About Pools

The Caller Pool contract maintains a lists of caller addresses. Whenver a change is made to the pool, either addition of a new member or removal of an existing member, a new pool is queued to take place of the current pool 512 blocks in the future. The new pool has all of the previous pool's members plus or minus whatever additions or removals take place.

During the first 256 blocks prior to a queued pool becoming active, additional members may choose to enter or leave. The state of the queued pool becomes frozen and cannot be changed starting at the 256 blocks leading up to the pool becoming active.

Once the queued pool becomes active, members are once again allowed to enter and exit the pool.

Each time a new pool is created, the ordering of its members is shuffled.

Note: It is worth pointing out that from the block during which you exit the pool, you must still execute the calls that are allocated to you for the next 512 blocks until the new queue becomes active. Failing to do so will cause bond forfeiture.

5.4 Entering the Pool

An address can enter the caller pool if the following conditions are met.

- The caller has deposited the minimum bond amount into their account with the Caller Pool.
- The caller is not in the active pool, or the next queued pool.

- The next queued pool does not go active within the next 256 blocks.

To enter the pool, call the `enterPool` function on the Caller Pool.

- **Solidity Function Signature:** `enterPool()`
- **ABI Signature:** `0x50a3bd39`

If the appropriate conditions are met, you will be added to the next caller pool. This will create a new pool if one has not already been created. Otherwise you will be added to the next queued pool.

You can use the `canEnterPool` function to check whether a given address is currently allowed to enter the pool.

- **Solidity Function Signature:** `canEnterPool(address callerAddress) returns (bool)`
- **ABI Signature:** `0x8dd5e298`

5.5 Exiting the Pool

An address can exit the caller pool if the following conditions are met.

- The caller is in the current active pool.
- The caller has not already exited or been removed from the queued pool (if it exists)
- The next queued pool does not go active within the next 256 blocks.

To exit the pool, use the `exitPool` function on the Caller Pool.

- **Solidity Function Signature:** `exitPool()`
- **ABI Signature:** `0x50a3bd39`

If all conditions are met, a new caller pool will be queued if one has not already been created and your address will be removed from it.

You can use the `canExitPool` function to check whether a given address is currently allowed to exit the pool.

- **Solidity Function Signature:** `canExitPool(address callerAddress) returns (bool)`
- **ABI Signature:** `0xb010d94a`

CHAPTER 6

Call Execution

Call execution is the process through which scheduled calls are executed at their desired block number. After a call has been scheduled, it can be executed by account which chooses to initiate the transaction. In exchange for executing the scheduled call, they are paid a small fee of approximately 1% of the gas cost used for executing the transaction.

6.1 Executing a call

Use the `doCall` function to execute a scheduled call.

- **Solidity Function Signature:** `doCall(bytes32 callKey)`
- **ABI Signature:** `0xfcf36918`

When this function is called, the following things happen.

1. A few are done to be sure that all of the necessary pre-conditions pass. If any fail, the function exits early without executing the scheduled call:
    - the scheduler has enough funds to pay for the execution.
    - the call has not already been executed.
    - the call has not been cancelled.
    - the current block number is within the range this call is allowed to be executed.
2. The necessary funds to pay for the call are put on hold.
3. The call is executed via either the **authorizedAddress** or **unauthorizedAddress** depending on whether the scheduler is an authorized caller.
4. The gas cost and fees are computed, deducted from the scheduler's account, and deposited in the caller's account.

6.1.1 Setting transaction gas and gas price

It is best to supply the maximum allowed gas when executing a scheduled call as the payment amount for executing the call is porportional to the amount of gas used. If the transaction runs out of gas, no payment is issued.

The payment is also dependent on the gas price for the executing transaction. The lower the gas price supplied, the higher the payment will be. (though you should make sure that the gas price is high enough that the transaction will get picked up by miners).

6.1.2 Getting your payment

Payment for executing a call is deposited in your Alarm service account and can be withdrawn using the account management api.

6.2 Determining what scheduled calls are next

The Alarm service uses Grove to facilitate querying for the next scheduled call.

- Grove Contract Address: `0xfe9d4e5717ec0e16f8301240df5c3f7d3e9effef`

6.2.1 The Grove Index

Grove tracks the ordering of data with indexes. You can retrieve the `bytes32` id one of two ways.

- From the Alarm service using the `getGroveIndexId()` function.
- From Grove using the `getIndexId(address ownerAddress ,bytes32 indexName)`

The index name used by Alarm is `'callTargetBlock'`.

TODO: put in grove address and alarm address here.

6.2.2 Querying Grove

- `query(bytes32 indexID, bytes2 operator, int value)`

One you have the index id, you will want to use the `query` function on Grove to get the first scheduled call after the current block.

```
> nodeId = grove.query.call(indexId, ">=", currentBlock)
```

This will return either `0x0` if there is no upcoming call, or a `bytes32` node id for the first node in the tree that matches our query. With the node id, we then need to fetch the value for the node id using the `getNodeValu(bytes32 nodeId)e` function.

```
> targetBlock = grove.getNodeValue.call(nodeId)
```

The return value represents the `targetBlock` value for the call. If we choose to execute this scheduled call when the block comes around, we need to have the `callKey`. We can retrieve this with the `getNodeId(bytes32 nodeId)` function on Grove since Alarm uses the `callKey` for each scheduled call as it's id in the grove index.

```
> callKey = grove.getNodeId.call(nodeId)
```

We should also check to see if there are more calls with the same target node. We can do this with the `getNextNode(bytes32 nodeId)` function on grove.

```
> next_node = grove.getNextNode.call(nodeId)
```

You can then repeate this process until the `targetBlock` is beyond the point in the future that you care to monitor.

Note: 40 blocks into the future is a good range to monitor since new calls must always be scheduled at least 40 blocks in the future.

6.2.3 The Grove Documentation

Detailed information about grove can be found in Grove's documentation.

6.3 Designated Callers

If the Caller Pool has any bonded callers in the current active pool, then only designated callers will be allowed to execute a scheduled call. The exception to this restriction is the last few blocks within the call's grace period which the call enters *free-for-all* mode during which anyone may execute it.

If there are no bonded callers in the Caller Pool then the Alarm service will operate in *free-for-all* mode for all calls meaning anyone may execute any call at any block during the call window.

6.3.1 How callers designated

Each call has a window during which it is allowed to be executed. This window begins at the specified `targetBlock` and extends through `targetBlock + gracePeriod`. This window is inclusive of it's bounding blocks.

For each 4 block section of the call window, the caller pool associated with the `targetBlock` is selected. The members of the pool can be though of as a circular queue, meaning that when you iterate through them, when you reach the last member, you start back over at the first member. For each call, a random starting position is selected in the member queue and the 4 block sections of the call window are assigned in order to the membes of the call pool beginning at this randomly chosen index..

The last two 4 block sections (5-8 blocks depending on the gracePeriod) are not allocated, but are considered *free-for-all* allowing anyone to call.

Use the `getDesignatedCaller` function to determine which caller from the caller pool has been designated for the block.

- **Solidity Function Signature:** `getDesignatedCaller(bytes32 callKey, uint targetBlock, uint8 gracePeriod, uint blockNumber) public returns (address)`
- **ABI Signature:** `0xe8543d0d`
- **callKey:** specifies the scheduled call.
- **targetBlock:** the target block for the specified call.
- **gracePeriod:** the grace period for the specified call.
- **blockNumber:** the block number (during the call window) in question.

This returns the address of the caller who is designated for this block, or `0x0` if this call can be executed by anyone on the specified block.

6.3.2 Missing the call window

Anytime a caller fails to execute a scheduled call during the 4 block window reserved for them, the next caller has the opportunity to claim a portion of their bond merely by executing the call during their window. When this happens, the previous caller who missed their call window has the current minimum bond amount deducted from their bond balance and transferred to the caller who executed the call. The caller who missed their call is also removed from the pool. This removal takes 512 blocks to take place as it occurs within the same mechanism as if they removed themselves from the pool.

6.3.3 Free For All

When a call enters the last two 4-block chunks of its call window it enters free-for-all mode. During these blocks anyone, even unbonded callers, can execute the call. The sender of the executing transaction will be rewarded the bond bonus from all callers who missed their call window.

6.4 Safeguards

There are a limited set of safeguards that Alarm protects those executing calls from.

- Enforces the ability to pay for the maximum possible transaction cost up front.
- Ensures that the call cannot cause the executing transaction to fail due to running out of gas (like an infinite loop).
- Ensures that the funds to be used for payment are locked during the call execution.

6.5 Tips for executing scheduled calls

The following tips may be useful if you wish to execute calls.

6.5.1 Only look in the next 40 blocks

Since calls cannot be scheduled less than 40 blocks in the future, you can count on the call ordering remaining static for the next 40 blocks.

6.5.2 No cancellation in next 8 blocks

Since calls cannot be cancelled less than 8 blocks in the future, you don't need to check cancellation status during the 8 blocks prior to its target block.

6.5.3 Check that it was not already called

If you are executing a call after the target block but before the grace period has run out, it is good to check that it has not already been called.

6.5.4 Check that the scheduler can pay

It is good to check that the scheduler has sufficient funds to pay for the call's potential gas cost plus fees.

CHAPTER 7

Call pricing and fees

The Alarm service operates under a **scheduler pays** model, which means that the scheduler of a call is responsible for paying for the full gas cost and fees associated with executing the call.

This payment is automatic and happens during the course of the execution of the scheduled call.

7.1 Minimum Balance

In order to guarantee reimbursment of gas costs and payment to the account which executes the scheduled call, the scheduler of the call must have an account balance sufficient to pay for the call at the time of execution. Since, it is unknown how much gas the call will consume the Alarm service requires a minimum balance equal to the maximum possible transaction cost plus fees.

7.2 Call Fees and Caller Payment

The account which executes the scheduled call is reimbursed 100% of the gas cost + payment for their service. The creator of the Alarm service is also paid the same payment.

The payment value is computed with the formula `1% of GasUsed * BaseGasPrice * GasPriceScalar` where:

- **GasUsed:** is the total gas consumption for the call execution. This includes all of the gas used by the Alarm service to do things like looking up call data, checking for sufficient account balance to pay for the call, paying the caller, etc.
- **BaseGasPrice** is the gas price that was used by the scheduler when they scheduled the function call.
- **GasPriceScalar** is a multiplier that ranges from 0 - 2 which is based on the difference between the gas priced used for call execution and the gas price used during call scheduling. This number incentivises the call executor to use as low a gas price as possible.

7.2.1 The GasPriceScalar multiplier

This multiplier is computed with the following formula.

- *IF* `gasPrice > baseGasPrice`

`baseGasPrice / gasPrice`

- *IF* `gasPrice <= baseGasPrice`

```
baseGasPrice / (2 * baseGasPrice - gasPrice)
```

Where:

- **baseGasPrice** is the `tx.gasprice` used when the call was scheduled.
- **gasPrice** is the `tx.gasprice` used to execute the call.

At the time of call execution, the `baseGasPrice` has already been set, so the only value that is variable is the `gasPrice` which is set by the account executing the transaction. Since the scheduler is the one who ends up paying for the actual gas cost, this multiplier is designed to incentivize the caller using the lowest gas price that can be expected to be reliably picked up and promptly executed by miners.

Here are the values this formula produces for a `baseGasPrice` of 20 and a `gasPrice` ranging from 10 - 40 which uses 5000 gas;

gasPrice	multiplier	payout
15	1.20	120
16	1.17	117
17	1.13	113
18	1.09	109
19	1.05	105
20	1.00	100
21	0.95	95
22	0.91	91
23	0.87	87
24	0.83	83
25	0.80	80
26	0.77	77
27	0.74	74
28	0.71	71
29	0.69	69
30	0.67	67
31	0.65	65
32	0.63	63
33	0.61	61
34	0.59	59
35	0.57	57
36	0.56	56
37	0.54	54
38	0.53	53
39	0.51	51
40	0.50	50

You can see from this table that as the `gasPrice` for the executing transaction increases, the total payout for executing the call decreases. This provides a strong incentive for the entity executing the transaction to use a reasonably low value.

Alternatively, if the `gasPrice` is set too low (potentially attempting to maximize payout) and the call is not picked up by miners in a reasonable amount of time, then the entity executing the call will not get paid at all. This provides a strong incentive to provide a value high enough to ensure the transaction will be executed.

7.3 Overhead

The gas overhead that you can expect to pay for your function call is approximately 146287.

CHAPTER 8

Contract ABI

Beyond the simplest use cases, the use of `address.call` to interact with the Alarm service is limiting. Beyond the readability issues, it is not possible to get the return values from function calls when using `call()`.

By using an abstract solidity contract which defines all of the function signatures, you can easily call any of the Alarm service's functions, letting the compiler handle computation of the function ABI signatures.

8.1 Abstract Solidity Contracts

The following abstract contracts can be used alongside your contract code to interact with the Alarm and CallerPool service.

8.1.1 Abstract Alarm Contract Source Code

```
contract AlarmAPI {
    /*
     *  Account Management API
     */
    function accountBalances(address account) public returns (uint);

    event Deposit(address indexed _from, address indexed accountAddress, uint value);
    function deposit(address accountAddress) public;

    event Withdraw(address indexed accountAddress, uint value);
    function withdraw(uint value) public;

    /*
     *  Authorization API
     */
    function unauthorizedAddress() public returns (address);
    function authorizedAddress() public returns (address);
    function addAuthorization(address schedulerAddress) public;
    function removeAuthorization(address schedulerAddress) public;
    function checkAuthorization(address schedulerAddress, address contractAddress) public returns (bo

    /*
     *  Scheduled Call Meta API
     */
    function getLastCallKey() public returns (bytes32);
    function getLastDataHash() public returns (bytes32);
```

```
    function getLastDataLength() public returns (uint);
    function getLastData() public returns (bytes);

    function getCallContractAddress(bytes32 callKey) public returns (address);
    function getCallScheduledBy(bytes32 callKey) public returns (address);
    function getCallCalledAtBlock(bytes32 callKey) public returns (uint);
    function getCallGracePeriod(bytes32 callKey) public returns (uint);
    function getCallTargetBlock(bytes32 callKey) public returns (uint);
    function getCallBaseGasPrice(bytes32 callKey) public returns (uint);
    function getCallGasPrice(bytes32 callKey) public returns (uint);
    function getCallGasUsed(bytes32 callKey) public returns (uint);
    function getCallABISignature(bytes32 callKey) public returns (bytes4);
    function checkIfCalled(bytes32 callKey) public returns (bool);
    function checkIfSuccess(bytes32 callKey) public returns (bool);
    function checkIfCancelled(bytes32 callKey) public returns (bool);
    function getCallDataHash(bytes32 callKey) public returns (bytes32);
    function getCallPayout(bytes32 callKey) public returns (uint);
    function getCallFee(bytes32 callKey) public returns (uint);
    function getCallData(bytes32 callKey) public returns (bytes);

    /*
     *  Call Data Registration API
     */
    event DataRegistered(bytes32 indexed dataHash);

    /*
     *  Call Scheduling API
     */
    event CallScheduled(bytes32 indexed callKey);
    event CallRejected(bytes32 indexed callKey, bytes12 reason);
    event CallCancelled(bytes32 indexed callKey);

    function getCallKey(address scheduledBy, address contractAddress, bytes4 abiSignature, bytes32 da
    function scheduleCall(address contractAddress, bytes4 abiSignature, bytes32 dataHash, uint target
    function scheduleCall(address contractAddress, bytes4 abiSignature, bytes32 dataHash, uint target
    function scheduleCall(address contractAddress, bytes4 abiSignature, bytes32 dataHash, uint target
    function cancelCall(bytes32 callKey) public;

    /*
     *  Grove data getters
     */
    function getGroveAddress() constant returns (address);
    function getGroveIndexName() constant returns (bytes32);
    function getGroveIndexId() constant returns (bytes32);

    /*
     *  Call Execution API
     */
    event CallExecuted(address indexed executedBy, bytes32 indexed callKey);
    event CallAborted(address indexed executedBy, bytes32 indexed callKey, bytes18 reason);

    function doCall(bytes32 callKey) public;
    function getCallMaxCost(bytes32 callKey) public returns (uint);
    function getCallFeeScalar(uint baseGasPrice, uint gasPrice) public returns (uint);

    function getCallerPoolAddress() public returns (address);
}
```

8.1.2 Register Data is special

You may notice that the contract above is missing the `registerData` function. This is because it is allowed to be called with any call signature and solidity has no way of defining such a function.

Registering your data requires use of the `address.call()` api.

8.1.3 Abstract CallerPool Contract Source Code

```
contract CallerPoolAPI {
    /*
     *  Bond managment API.
     */
    function callerBonds(address callerAddress) public returns (uint);
    function getMinimumBond() public returns (uint);
    function depositBond() public;
    function withdrawBond(uint value) public;

    /*
     *  CallerPool <=> Alarm api.
     */
    function getDesignatedCaller(bytes32 callKey, uint targetBlock, uint8 gracePeriod, uint blockNumb

    event AwardedMissedBlockBonus(address indexed fromCaller, address indexed toCaller, uint indexed

    /*
     *  Pool querying
     */
    function poolHistory(uint index) returns (uint);
    function getPoolKeyForBlock(uint blockNumber) public returns (uint);
    function getActivePoolKey() public returns (uint);
    function getNextPoolKey() public returns (uint);
    function getPoolSize(uint poolKey) constant returns (uint);
    function getPoolFreezeDuration() constant returns (uint);
    function getPoolMinimumLength() constant returns (uint);

    /*
     *  Pool membership API
     */
    function isInAnyPool(address callerAddress) public returns (bool);
    function isInPool(address callerAddress, uint poolNumber) public returns (bool);

    /*
     *  Enter/Exit pool API
     */
    function canEnterPool(address callerAddress) public returns (bool);
    function canExitPool(address callerAddress) public returns (bool);
    function enterPool() public;
    function exitPool() public;
}
```

8.1.4 Only use what you need

The contracts above have stub functions for every API exposed by Alarm and CallerPool. It is safe to remove any functions or events from the abstract contracts that you do not intend to use.

CHAPTER 9

Caller Pool API

The Caller Pool contract exposes the following api functions.

9.1 Bond Management

The following functions are available for managing the ether deposited as a bond with the Caller Pool.

9.1.1 Get Minimum Bond

Use the `getMinimumBond` function to retrieve the current minimum bond value required to be able to enter the caller pool.

- **Solidity Function Signature:** `getMinimumBond() returns (uint)`
- **ABI Signature:** `0x23306ed6`

9.1.2 Check Bond Balance

Use the `callerBonds` function to check the bond balance for the provided address.

- **Solidity Function Signature:** `callerBonds(address callerAddress) returns (uint)`
- **ABI Signature:** `0xc861cd66`

9.1.3 Deposit Bond

Use the `depositBond` function to deposit you bond with the caller pool.

- **Solidity Function Signature:** `depositBond()`
- **ABI Signature:** `0x741b3c39`

9.1.4 Withdraw Bond

Use the `withdrawBond` function to withdraw funds from your bond.

- **Solidity Function Signature:** `withdrawBond()`
- **ABI Signature:** `0xc3daab96`

When in either an active or queued caller pool, you cannot withdraw your account below the minimum bond value.

9.2 Call Scheduling and Execution

The following function is available for callers.

9.2.1 Get Designated Caller

Use the `getDesignatedCaller` function to retrieve which caller address, if any, is designated as the caller for a given block and scheduled call.

- **Solidity Function Signature:** `getDesignatedCaller(bytes32 callKey, uint targetBlock, uint8 gracePeriod, uint blockNumber) public returns (address)`
- **ABI Signature:** `0xe8543d0d`
- **callKey:** specifies the scheduled call.
- **targetBlock:** the target block for the specified call.
- **gracePeriod:** the grace period for the specified call.
- **blockNumber:** the block number (during the call window) in question.

This returns the address of the caller who is designated for this block, or `0x0` if this call can be executed by anyone on the specified block.

9.3 Pool Information

The following functions are available to query information about call pools.

9.3.1 Pool History

Use the `poolHistory` function to lookup historical caller pools.

- **Solidity Function Signature:** `poolHistory(uint index) returns (uint)`
- **ABI Signature:** `0x910789c4`

This function can be used to return the nth caller pool, where index is the 0-indexed number of the desired caller pool. Returns the `poolKey` which can be used to reference the caller pool. The `poolKey` is also the block number that the pool became active.

9.3.2 Get Pool Key for Block

Use the `getPoolKeyForBlock` function to return the `poolKey` that should be used for the given block number.

- **Solidity Function Signature:** `getPoolKeyForBlock(uint blockNumber) returns (uint)`
- **ABI Signature:** `0xaec918c7`

9.3.3 Get Active Pool Key

Use the `getActivePoolKey` function to retrieve the `poolKey` for the caller pool that is currently active.

- **Solidity Function Signature:** `getActivePoolKey() returns (uint)`
- **ABI Signature:** `0xa6814e8e`

9.3.4 Get Next Pool Key

Use the `getNextPoolKey` function to retrieve the `poolKey` that is currently queued up next.

- **Solidity Function Signature:** `getNextPoolKey() returns (uint)`
- **ABI Signature:** `0xc4afc3fb`

Returns `0` if there is no caller pool queued.

9.3.5 Get Pool Size

Use the `getPoolSize` function to lookup the size of a given pool.

- **Solidity Function Signature:** `getPoolSize(uint poolKey) returns (uint)`
- **ABI Signature:** `0x6595f73a`

9.4 Pool Membership

The following functions can be used to query about an address's pool membership.

9.4.1 Is In Any Pool

Use the `isInAnyPool` function to query whether an address is in either the currently active caller pool or the queued caller pool.

- **Solidity Function Signature:** `isInAnyPool(address callerAddress) returns (bool)`
- **ABI Signature:** `0x84c92c9a`

9.4.2 Is In Pool

Use the `isInPool` function to query whether an address is in a specific pool.

- **Solidity Function Signature:** `isInPool(address callerAddress, uint poolKey) returns (bool)`
- **ABI Signature:** `0x19f74e1f`

9.5 Entering and Exiting Pools

The following functions can be used for actions related to entering and exiting the call pool.

9.5.1 Can Enter Pool

Use the `canEnterPool` function to query whether or not you are allowed to enter the caller pool.

- **Solidity Function Signature:** `canEnterPool() returns (bool)`
- **ABI Signature:** `0x8dd5e298`

9.5.2 Can Exit Pool

Use the `canExitPool` function to query whether or not you are allowed to exit the caller pool.

- **Solidity Function Signature:** `canExitPool() returns (bool)`
- **ABI Signature:** `0xb010d94a`

9.5.3 Enter Pool

Use the `enterPool` function to enter the caller pool.

- **Solidity Function Signature:** `enterPool() returns (bool)`
- **ABI Signature:** `0x50a3bd39`

9.5.4 Exit Pool

Use the `exitPool` function to exit the caller pool.

- **Solidity Function Signature:** `exitPool() returns (bool)`
- **ABI Signature:** `0x29917954`

CHAPTER 10

Scheduled Call API

The Alarm service exposes getter functions for all call information that may be important to those scheduling or executing calls.

10.1 Properties of a Scheduled Call

- **bytes32 callKey:** the unique identifier for this function call.
- **address contractAddress:** the address of the contract the function should be called on.
- **address scheduledBy:** the address who scheduled the call.
- **uint calledAtBlock:** the block number on which the function was called. (`0` if the call has not yet been executed.)
- **uint targetBlock:** the block that the function should be called on.
- **uint8 gracePeriod:** the number of blocks after the `targetBlock` during which it is stll ok to execute the call.
- **uint nonce:** value to differentiate multiple *identical* calls that should happen simultaneously.
- **uint baseGasPrice:** the gas price that was used when the call was scheduled.
- **uint gasPrice:** the gas price that was used when the call was executed. (`0` if the call has not yet been executed.)
- **uint gasUsed:** the amount of gas that was used to execute the function call (`0` if the call has not yet been executed.)
- **uint payout:** the amount in wei that was paid to the address that executed the function call. (`0` if the call has not yet been executed.)
- **uint fee:** the amount in wei that was kept to pay the creator of the Alarm service. (`0` if the call has not yet been executed.)
- **bytes4 sig:** the 4 byte ABI function signature of the function on the `contractAddress` for this call.
- **bool isCancelled:** whether the call was cancelled.
- **bool wasCalled:** whether the call was called.
- **bool wasSuccessful:** whether the call was successful.
- **bytes32 dataHash:** the `sha3` hash of the data that should be used for this call.

10.1.1 Call Key

bytes32 callKey

The following functions are available on the Alarm service. The vast majority of them take the **callKey** which is an identifier used to reference a scheduled call.

The **callKey** is computed as `sha3(scheduledBy, contractAddress, signature, dataHash, targetBlock, gracePeriod, nonce)` where:

- **scheduledBy:** the `address` that scheduled the call.
- **contractAddress:** the `address` of the contract that the function should be called on when this call is executed.
- **signature:** the `byte4` ABI function signature of the function that should be called.
- **dataHash:** the `bytes32` sha3 hash of the call data that should be used for this scheduled call.
- **targetBlock:** the `uint256` block number that this call should be executed on.
- **gracePeriod:** the `uint8` number of blocks after `targetBlock` during which it is still ok to execute this scheduled call.
- **nonce:** the `uint256` value that can be used to distinguish between multiple calls with identical data that should occur during the same time. This value only matters if you are registering multiple calls for which all of the other fields are the same.

10.1.2 Contract Address

address contractAddress

The address of the contract that the scheduled function call should be executed on. Retrieved with the `getCallContractAddress` function.

- **Solidity Function Signature:** `getCallContractAddress(bytes32 callKey) returns (address)`
- **ABI Signature:** `0x9c975df`

10.1.3 Scheduled By

address scheduledBy

The address of the contract that the scheduled function call should be executed on. Retrieved with the `getCallScheduledBy` function.

- **Solidity Function Signature:** `getCallScheduledBy(bytes32 callKey) returns (address)`
- **ABI Signature:** `0x8b37e656`

10.1.4 Called at Block

uint calledAtBlock

The block number that this call was executed. Retrieved with the `getCallCalledAtBlock` function. Returns `0` if the call has not been executed yet.

- **Solidity Function Signature:** `getCallCalledAtBlock(bytes32 callKey) returns (uint)`
- **ABI Signature:** `0xe4098655`

10.1.5 Grace Period

uint8 gracePeriod

The number of blocks after the `targetBlock` that it is still ok to execute this call. Retrieved with the `getCallGracePeriod` function.

- **Solidity Function Signature:** `getCallGracePeriod(bytes32 callKey) returns (uint8)`
- **ABI Signature:** `0x34c19b93`

10.1.6 Target Block

uint targetBlock

The block number that this call should be executed on. Retrieved with the `getCallTargetBlock` function.

- **Solidity Function Signature:** `getCallTargetBlock(bytes32 callKey) returns (uint)`
- **ABI Signature:** `0x234917d4`

10.1.7 Base Gas Price

uint baseGasPrice

The value of `tx.gasprice` that was used to schedule this function call. Retrieved with the `getCallBaseGasPrice` function. Returns `0` if the call has not been executed yet.

- **Solidity Function Signature:** `getCallBaseGasPrice(bytes32 callKey) returns (uint)`
- **ABI Signature:** `0x77b19cd5`

10.1.8 Gas Price

uint gasPrice

The value of `tx.gasprice` that was used to execute this function call. Retrieved with the `getCallGasPrice` function. Returns `0` if the call has not been executed yet.

- **Solidity Function Signature:** `getCallGasPrice(bytes32 callKey) returns (uint)`
- **ABI Signature:** `0x78bc6460`

10.1.9 Gas Used

uint gasUsed

The amount of gas that was used during execution of this function call. Retrieved with the `getCallGasUsed` function. Returns `0` if the call has not been executed yet.

- **Solidity Function Signature:** `getCallGasUsed(bytes32 callKey) returns (uint)`
- **ABI Signature:** `0x86ae9e4`

10.1.10 Signature

bytes4 signature

The ABI function signature that should be used to execute this function call. Retrieved with the `getCallSignature` function.

- **Solidity Function Signature:** `getCallSignature(bytes32 callKey) returns (uint)`
- **ABI Signature:** `0xc88edaed`

10.1.11 Was Called

bool wasCalled

Boolean flag for whether or not this function has been called yet. Retrieved with the `checkIfCalled` function.

- **Solidity Function Signature:** `checkIfCalled(bytes32 callKey) returns (bool)`
- **ABI Signature:** `0x2a472ae8`

10.1.12 Was Successful

bool wasSuccessful

Boolean flag for whether or not this function call was successful when executed. Retrieved with the `checkIfSuccess` function.

- **Solidity Function Signature:** `checkIfSuccess(bytes32 callKey) returns (bool)`
- **ABI Signature:** `0x6ffc0896`

10.1.13 Is Cancelled

bool isCancelled

Boolean flag for whether or not this function call was cancelled. Retrieved with the `checkIfCancelled` function.

- **Solidity Function Signature:** `checkIfCancelled(bytes32 callKey) returns (bool)`
- **ABI Signature:** `0xaa4cc01f`

10.1.14 Call Data Hash

bytes32 dataHash

The sha3 hash of the call data that will be used for this function call. Retrieved with the `getCallDataHash` function.

- **Solidity Function Signature:** `getCallDataHash(bytes32 callKey) returns (bytes32)`
- **ABI Signature:** `0xf9f447eb`

10.1.15 Call Data

bytes data

The full call data that will be used for this function call. Retrieved with the `getCallData` function.

- **Solidity Function Signature:** `getCallData(bytes32 callKey) returns (bytes)`
- **ABI Signature:** `0x75428615`

10.1.16 Payout

uint payout

The amount in wei that was paid to the account that executed this function call. Retrieved with the `getCallPayout` function. If the function has not been executed this will return `0`.

- **Solidity Function Signature:** `getCallPayout(bytes32 callKey) returns (uint)`
- **ABI Signature:** `0xa9743c68`

10.1.17 Fee

uint fee

The amount in wei that was paid to the creator of the Alarm service. Retrieved with the `getCallFee` function. If the function has not been executed this will return `0`.

- **Solidity Function Signature:** `getCallFee(bytes32 callKey) returns (uint)`
- **ABI Signature:** `0xfc300522`

CHAPTER 11

Events

The following events are used to log notable events within the Alarm service.

11.1 Alarm Events

The primary Alarm service contract logs the following events.

11.1.1 Deposit

- **Solidity Event Signature:** `Deposit(address indexed _from, address indexed accountAddress, uint value)`
- **ABI Signature:** `0x5548c837`

Executed anytime a deposit is made into an address's Alarm account.

11.1.2 Withdraw

- **Solidity Event Signature:** `Withdraw(address indexed accountAddress, uint value)`
- **ABI Signature:** `0x884edad9`

Executed anytime a withdrawl is made from an address's Alarm account.

11.1.3 Call Scheduled

- **Solidity Event Signature:** `CallScheduled(bytes32 indexed callKey)`
- **ABI Signature:** `0x5ca1bad5`

Executed when a new scheduled call is created.

11.1.4 Call Executed

- **Solidity Event Signature:** `CallExecuted(address indexed executedBy, bytes32 indexed callKey)`
- **ABI Signature:** `0xed1062ba`

Executed when a scheduled call is executed.

11.1.5 Call Aborted

- **Solidity Event Signature:** `CallAborted(address indexed executedBy, bytes32 indexed callKey, bytes18 reason)`
- **ABI Signature:** `0x84b46e45`

Executed when an attempt is made to execute a scheduled call is rejected. The `reason` value in this log entry contains a short string representation of why the call was rejected.

11.2 Caller Pool Events

The Caller Pool contract logs the following events.

11.2.1 Added To Pool

- **Solidity Event Signature:** `AddedToPool(address indexed callerAddress, uint indexed pool)`
- **ABI Signature:** `0xa192e48a`

Executed anytime a new address is added to the caller pool.

11.2.2 Removed From Pool

- **Solidity Event Signature:** `RemovedFromPool(address indexed callerAddress, uint indexed pool)`
- **ABI Signature:** `0xeee53013`

Executed anytime an address is removed from the caller pool.

11.2.3 Awarded Missed Block Bonus

- **Solidity Event Signature:** `AwardedMissedBlockBonus(address indexed fromCaller, address indexed toCaller, uint indexed poolNumber, bytes32 callKey, uint blockNumber, uint bonusAmount)`
- **ABI Signature:** `0x47d4e871`

Executed anytime a pool member's bond is awarded to another address due to them missing a scheduled call that was designated as theirs to execute.

CHAPTER 12

Changelog

12.1 0.3.0

- Convert Alarm service to use Grove for tracking scheduled call ordering.
- Enable logging most notable Alarm service events.
- Two additional convenience functions for invoking `scheduleCall` with **gracePeriod** and **nonce** as optional parameters.

12.2 0.2.0

- Fix for Issue 42. Make the free-for-all bond bonus restrict itself to the correct set of callers.
- Re-enable the right tree rotation in favor of removing three `getLastX` function. This is related to the pi-million gas limit which is restricting the code size of the contract.

12.3 0.1.0

- Initial release.