Ethereum Alarm Clock Documentation

Release 1.0.0

Piper Merriam

December 23, 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 executed by other nodes on the ethereum network in exchange for a small payment and reimbursment of gas costs.

The service is completely trustless, open source, and verifiable.

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 zero trust between any of the users of the service, as well as providing no special access to any party (including the author of the service)

1.1 Scheduling Function Calls

Contracts, or individuals can schedule a function calls with the Alarm service by doing the following.

1. Schedule the function call with the service, providing basic information such as what function to call and when it should be called.
2. Register any call data that will be required to make the function call (optional for functions that have no arguments).

1.1.1 Call Scheduling

Function calls can be scheduled for any block at least 305 blocks *(~75 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. Target block number that the call should be executed on.

Optionally, these additional pieces of information can be supplied.

- Suggested gas amount that should be provided to the function. **default: 0 to indicate no suggestion**
- Number of blocks after the target block during which it still ok to execute the call. (between 64 - 255 blocks) **default: 255**
- Payment amount in wei that will be paid to the executor of the call. **default: 1 ether**
- Fee amount in wei that will be paid to the creator of the Alarm service. **default: 100 finey**

The scheduling transaction must also include enough ether to pay for the gas costs of the call as well as the payment and fee values.

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

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.

Note: Functions which do not have any arguments can skip this step.

1.2 Execution of scheduled calls

Scheduled function calls can be executed by anyone who wishes to initiate the transaction. Each call has a payment amount associated with it which is paid to the executor of the call.

1.2.1 Cost

In addition to the gas costs, schedulers are also encouraged to pay the call executor as well as the creator of the service for their effort. This value can be specified by the scheduler, meaning that you may choose to offer any amount for the execution of your function..

The scheduling function uses the following defaults if specific values are not provided.

- Payment to the executor: **1 ether**
- Payment to the service creator: **100 finney**

1.3 Guarantees

1.3.1 Will the call happen?

There are no guarantees that your function will be called. This is not a shortcoming of the service, but rather a fundamental limitation of the ethereum network. Nobody is capable of forcing a transaction to be included in a specific block.

The Alarm service has been designed such that it **should** become more reliable as more people use it.

However, it is entirely possible that certain calls will be missed due to unforseen circumstances. Providing a higher Payment amount is a potential way to get your scheduled call handled at a higher priority as it will be more profitable to execute.

1.3.2 Will I get paid for executing a call?

If you are diligent about how you go about executing scheduled calls then executing scheduled calls is guaranteed to be profitable. See the section on executing calls for more information.

CHAPTER 2

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.

When a call is scheduled, the service deploys a new contract that represents the scheduled function call. This contract is referred to as the **call contract**. It holds all of the metadata associated with the call as well as the funds that will be used to pay for the call.

2.1 Lifecycle of a Call Contract

- **creation** - The call contract is created as part of call scheduling.
- **data-registered** - The data for the call is registered with the contract. This step can be skipped for function calls that take no arguments.
- **locked** - The contract is locked, preventing cancellation starting 10 blocks before the call's target block through the last block in the call window.
- **execution** - The executing transaction is sent, which triggers the call contract to execute the function call.
- **payment** - payments are sent to the executor and the creator of the alarm service.
- **suicide** - The contract suicides itself, sending the remainder of it's funds to the call scheduler.

2.2 Scheduling the Call

Function calls are scheduled with the `scheduleCall` function on the Alarm service. This creates a new **call contract** that represents the function call.

- **Solidity Function Signature:** `function scheduleCall(address contractAddress, bytes4 abiSignature, uint targetBlock, uint suggestedGas, uint8 gracePeriod, uint basePayment, uint baseFee) public returns (address)`
- **ABI Signature:** `0x8b676ae8`

The `scheduleCall` function takes the following parameters:

Required Arguments

- **address contractAddress:** The address of the contract for the function call.
- **bytes4 abiSignature:** The 4 byte ABI signature of the function to be called.

- **uint targetBlock:** The block number the call should be executed on.

Optional Arguments

- **uint suggestedGas:** A suggestion to the call executor as to how much gas should be provided to execute the call. (default: 0)
- **uint8 gracePeriod:** The number of blocks after `targetBlock` that it is ok to still execute this call. Cannot be less than 64. (default: 255)
- **uint basePayment:** The base amount in wei that should be paid to the executor of the call. (default: 1 ether)
- **uint baseFee:** The base amount in wei that should be paid to the creator of the alarm service. (default: 100 finney)

The optional arguments are implemented through the following alternate invocation signatures. The default value for each of the optional arguments will be used if any of the following signatures are used.

- **Solidity Function Signature:** `function scheduleCall(address contractAddress, bytes4 abiSignature, uint targetBlock) public returns (address)`
- **ABI Signature:** `0x1991313`
- **Solidity Function Signature:** `function scheduleCall(address contractAddress, bytes4 abiSignature, uint targetBlock, uint suggestedGas) public returns (address)`
- **ABI Signature:** `0x49ae734`
- **Solidity Function Signature:** `function scheduleCall(address contractAddress, bytes4 abiSignature, uint targetBlock, uint suggestedGas, uint8 gracePeriod) public returns (address)`
- **ABI Signature:** `0x480b70bd`
- **Solidity Function Signature:** `function scheduleCall(address contractAddress, bytes4 abiSignature, uint targetBlock, uint suggestedGas, uint8 gracePeriod, uint basePayment) public returns (address)`
- **ABI Signature:** `0x68402460`

If the `scheduleCall` function is being used from within a contract, the address of the newly created call contract is returned. If instead, the function is being called directly in a transaction, the address of the call contract can be extracted from the transaction logs under the `CallScheduled` event.

2.2.1 Contract scheduling its own call

Contracts can take care of their own call scheduling.

```
contract Lottery {
    address alarm; // set by some other mechanism.

    function beginLottery() public {
        ... // Do whatever setup needs to take place.

        // Now we schedule the picking of the winner.

        bytes4 sig = bytes4(sha3("pickWinner()"));
        // approximately 24 hours from now
        uint targetBlock = block.number + 5760;
        // 0x1991313 is the ABI signature computed from `bytes4(sha3("scheduleCall(...)"))`.
        alarm.call(0x1991313, address(this), sig, targetBlock)
```

```
    }

    function pickWinner() public {
        ...
    }
}
```

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).

2.2.2 Scheduling a call for a contract

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

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.

```
// Now schedule the call
> 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, targetBlock, {from: eth.coinbase, valu
```

2.3 Registering Call Data

If a function call requires arguments then it is up to the scheduler to register the call data. This needs to be done prior to execution.

The call contract allows for call data registration via two mechanisms. The primary mechanism is through the fallback function on the contract. This will set the call data as the full call data of the transaction.

```
// Register some call data
> web3.eth.sendTransaction({to: scheduler.address, data: "0x...."})
```

Or, from within your contract.

```
contract Lottery {
    address alarm; // set by some other mechanism.

    function beginLottery() public {
        uint lotteryId = ...;

        // Now we schedule the picking of the winner.
        bytes4 sig = bytes4(sha3("pickWinner(uint256)"));
        // 0x1991313 is the ABI signature computed from `bytes4(sha3("scheduleCall(address,bytes4,ui
        alarm.call(0x1991313, address(this), sig, 100)

        // Register the call data
        alarm.call(lotteryId);
    }

    function pickWinner(uint lotteryId) public {
        ...
    }
}
```

If however, your call data either has a `bytes4` value as it's first argument, or, the first 4 bytes of the call data have a collision with one of the existing function signatures on the call contract, you can use the `registerData` function instead.

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

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.

Once data has been registered, it cannot be modified. Attempts to do so will result in an exception.

2.3.1 ABI Encoding and address.call

The `call()` function on an address in solidity does not do any ABI encoding, so in cases where a scheduled call must pass something like a `bytes` variable, you will need to handle the ABI encoding yourself.

2.4 Cancelling a call

A scheduled call can be cancelled by its scheduler up to 10 blocks before it's target block. To cancel a scheduled call use the `cancel` function.

- **Solidity Function Signature:** `cancel()`
- **ABI Signature:** `0xea8a1af0`

This will cause the call to be set as **cancelled**, which will return any funds currently being held by the contract.

2.5 Looking up a Call

You can lookup whether a particular address is a known scheduled call with the `isKnownCall` function.

- **Solidity Function Signature:** `isKnownCall(address callAddress) returns (bool)`
- **ABI Signature:** `0x523ccfa8`

Returns a boolean as to whether this address represents a known scheduled call.

CHAPTER 3

Call Contract API

3.1 Properties of a Call Contract

A call contract for a scheduled call has the following publicly accessible values.

- **address contractAddress:** the address of the contract the function should be called on.
- **address schedulerAddress:** the address who scheduled the call.
- **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 anchorGasPrice:** the gas price that was used when the call was scheduled.
- **uint suggestedGas:** a suggestion to the call executor as to how much gas the called function is expected to need.
- **uint basePayment:** the amount in wei that will be paid to the address that executes the function call.
- **uint baseFee:** the amount in wei that will be paid the creator of the Alarm service.
- **bytes4 abiSignature:** the 4 byte ABI function signature of the function on the `contractAddress` for this call.
- **bytes callData:** the data that will be passed to the called function.
- **bool wasCalled:** whether the call was called.
- **bool wasSuccessful:** whether the call was successful during execution.
- **bool isCancelled:** whether the call was cancelled.
- **address claimer:** the address that has claimed this contract.
- **uint claimAmount:** the amount that the claimer agreed to execute the contract for.
- **uint claimerDeposit:** the amount that the claimer has put up for deposit.

3.1.1 Contract Address

address contractAddress

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

- **Solidity Function Signature:** `contractAddress() returns (address)`
- **ABI Signature:** `0xf6b4dfb4`

3.1.2 Scheduler Address

address schedulerAddress

The address that the scheduled function call. Retrieved with the `schedulerAddress` function.

- **Solidity Function Signature:** `schedulerAddress() returns (address)`
- **ABI Signature:** `0xae45850b`

3.1.3 Target Block

uint targetBlock

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

- **Solidity Function Signature:** `targetBlock() returns (uint)`
- **ABI Signature:** `0xa16697a`

3.1.4 Grace Period

uint8 gracePeriod

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

- **Solidity Function Signature:** `gracePeriod() returns (uint8)`
- **ABI Signature:** `0xa06db7dc`

3.1.5 Anchor Gas Price

uint anchorGasPrice

The value of `tx.gasprice` that was used to schedule this function call. Retrieved with the `anchorGasPrice` function.

- **Solidity Function Signature:** `anchorGasPrice() returns (uint)`
- **ABI Signature:** `0x37f4c00e`

3.1.6 Suggested Gas

uint suggestedGas

A suggestion for the amount of gas that a caller should expect the called function to require. Retrieved with the `suggestedGas` function.

- **Solidity Function Signature:** `suggestedGas() returns (uint)`
- **ABI Signature:** `0x6560a307`

3.1.7 Base Payment

uint basePayment

The base amount, in wei that the call executor's payment will be calculated from. Retrieved with the `basePayment` function.

- **Solidity Function Signature:** `basePayment() returns (uint)`
- **ABI Signature:** `0xc6502da8`

3.1.8 Base Fee

uint baseFee

The base amount, in wei that the fee to the creator of the alarm service will be calculate from. Retrieved with the `baseFee` function.

- **Solidity Function Signature:** `baseFee() returns (uint)`
- **ABI Signature:** `0x6ef25c3a`

3.1.9 ABI Signature

bytes4 abiSignature

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

- **Solidity Function Signature:** `abiSignature() returns (uint)`
- **ABI Signature:** `0xca94692d`

3.1.10 Call Data

bytes callData

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

- **Solidity Function Signature:** `callData() returns (bytes)`
- **ABI Signature:** `0x4e417a98`

3.1.11 Was Called

bool wasCalled

Boolean as to whether this call has been executed. Retrieved with the `wasCalled` function.

- **Solidity Function Signature:** `wasCalled() returns (bool)`
- **ABI Signature:** `0xc6803622`

3.1.12 Was Successful

bool wasSuccessful

Boolean as to whether this call was successful. This indicates whether the called contract returned without error. Retrieved with the `wasSuccessful` function.

- **Solidity Function Signature:** `wasSuccessful() returns (bool)`
- **ABI Signature:** `0x9241200`

3.1.13 Is Cancelled

bool isCancelled

Boolean as to whether this call has been cancelled. Retrieved with the `isCancelled` function.

- **Solidity Function Signature:** `isCancelled() returns (bool)`
- **ABI Signature:** `0x95ee1221`

3.1.14 Claimer

address claimer

Address of the account that has claimed this call for execution. Retrieved with the `claimer` function.

- **Solidity Function Signature:** `claimer() returns (address)`
- **ABI Signature:** `0xd379be23`

3.1.15 Claim Amount

uint claimAmount

Ammount that the `claimer` has agreed to pay for the call. Retrieved with the with the `claimAmount` function.

- **Solidity Function Signature:** `claimAmount() returns (uint)`
- **ABI Signature:** `0x830953ab`

3.1.16 Claim Deposit

uint claimerDeposit

Ammount that the `claimer` put down as a deposit. Retrieved with the with the `claimerDeposit` function.

- **Solidity Function Signature:** `claimerDeposit() returns (uint)`
- **ABI Signature:** `0x3233c686`

3.2 Functions of a Call Contract

3.2.1 Cancel

Cancels the scheduled call, suiciding the call contract and sending any funds to the scheduler's address. This function cannot be called from 265 blocks prior to the **target block** for the call through the end of the grace period.

Before the call, only the scheduler may cancel the call. Afterwards, anyone may cancel it.

- **Solidity Function Signature:** `cancel() public`
- **ABI Signature:** `0xea8a1af0`

3.2.2 Execute

Triggers the execution of the call. This can only be done during the window between the `targetBlock` through the end of the `gracePeriod`. If the call has been claimed, then only the claiming address can execute the call during the first 16 blocks. If the claming address does not execute the call during this time, anyone who subsequently executes the call will receive their deposit.

- **Solidity Function Signature:** `execute() public`
- **ABI Signature:** `0x61461954`

CHAPTER 4

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.

4.1 Executing a call

Use the `execute` function to execute a scheduled call. This function is present on the call contract itself (as opposed to the scheduling service).

- **Solidity Function Signature:** `execute() public`
- **ABI Signature:** `0x61461954`

When this function is called, the following things happen.

1. A few checks 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 call has not already been called.
    - the current block number is within the range this call is allowed to be executed.
    - the caller is allowed to execute the function (see caller pool)
2. The call is executed
3. The gas cost and fees are computed and paid.
4. The call contract sends any remaining funds to the scheduling address.

4.1.1 Payment

Each scheduled call sets its own payment value. This can be looked up with the `basePayment` accessor function.

The final payment value for executing the scheduled call is the `basePayment` multiplied by a scalar value based on the difference between the gas price of the executing transaction and the gas price that was used to schedule the transaction. The formula for this scalar is such that the lower the gas price of the executing transaction, the higher the payment.

4.1.2 Setting transaction gas and gas price

Each call contract has a `suggestedGas` property that can be used as a suggestion for how much gas the function call needs. In the case where this is set to zero it means the scheduler has not provided a suggestion.

This suggested gas value should be used in conjuction with the `basePayment` and `baseFee` amounts with respect to the ether balance of the call contract. The provided gas for the transaction should not exceed `(balance - 2 * (basePayment + baseFee)) / gasPrice` if you want to guarantee that you will be fully reimbursed for gas expenditures.

4.1.3 Getting your payment

Payment for executing a call is sent to you as part of the executing transaction, as well as being logged by the `CallExecuted` event.

4.2 Determining what scheduled calls are next

You can query the Alarm service for the call key of the next scheduled call on or after a specified block number using the `getNextCall` function

- **Solidity Function Signature:** `getNextCall(uint blockNumber) returns (address)`
- **ABI Signature:** `0x9f927be7`

Since there may be multiple calls on the same block, it is best to also check if the call has any *siblings* using the `getNextCallSibling` function. This function takes a call contract address and returns the address that is scheduled to come next.

When checking for additional calls in this manner, you should check the target block of each subsequent call to be sure it is within a range that you care about.

- **Solidity Function Signature:** `getNextCallSibling(address callAddress) returns (address)`
- **ABI Signature:** `0x48107843`

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. You should also monitor these functions up to 10 blocks before their target block to be sure they are not cancelled.

4.3 The Freeze Window

The 10 blocks prior to a call's target block are called the **freeze window**. During this window, nothing about a call can change. This means that it cannot be cancelled or claimed.

4.4 Claiming a call

Claiming a call is the process through which you as a call executor can guarantee the exclusive right to execute the call during the first 16 blocks of the call window for the scheduled call. As part of the claim, you will need to put down a deposit, which is returned to you if you when you execute the call. Failing to execute the call will forfeit your deposit.

4.4.1 Claim Amount

A call can be claimed during the 255 blocks prior to the freeze window. This period is referred to as the claim window. The amount that you are agreeing to be paid for the call is based on whichever block the call is claimed on. The amount can be calculated using the following formula.

- Let `i` be the index of the block within the 255 block claim window.
- Let `basePayment` be the payment amount specified by the call contract.
- If within the first 240 blocks of the window: `payment = basePayment * i / 240`
- If within the last 15 blocks of the window: `payment = basePayment`

This formula results in a linear growth from 0 to the full `basePayment` amount over the course of the first 240 blocks in the claim window. The last 15 blocks are all set at the full `basePayment` amount.

A claim must be accompainied by a deposit that is at least twice the call's `basePayment` amount.

4.4.2 Getting your Deposit Back

If you claim a call and do not execute it within the first 16 blocks of the call window, then you will risk losing your deposit. Once the first 16 blocks have passed, the call can be executed by anyone. At this point, the first person to execute the call will receive the deposit as part of their payment (and incentive to pick up claimed calls that have not been called).

4.4.3 Claim API

To claim a contract

- **Solidity Function Signature:** `claim()`
- **ABI Signature:** `0x4e71d92d`

To check what the `claimAmount` will be for a given block number use the `getClaimAmountForBlock` function. This will return an amount in wei that represents the base payment value for the call if claimed on that block.

- **Solidity Function Signature:** `getClaimAmountForBlock(uint blockNumber)`
- **ABI Signature:** `0xf5562753`

This function also has a shortcut that uses the current block number

- **Solidity Function Signature:** `getClaimAmountForBlock()`
- **ABI Signature:** `0x4f059a43`

You can check if a call has already been claimed with the `claimer` function. This function will return either the empty address `0x0` if the call has not been claimed, or the address of the claimer if it has.

- **Solidity Function Signature:** `claimer() returns (address)`
- **ABI Signature:** `0xd379be23`

4.5 Safeguards

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

- 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.

4.6 Tips for executing scheduled calls

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

4.6.1 Only look in the next 265 blocks

Since calls cannot be scheduled less than 265 blocks in the future, you can count on the call ordering remaining static for the next 265 blocks.

4.6.2 No cancellation in next 265 blocks

Since calls cannot be cancelled less than 265 blocks in the future, you don't need to check cancellation status during the 265 blocks prior to its target block.

4.6.3 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.

4.6.4 Compute how much gas to provide

If you want to guarantee that you will be 100% reimbursed for your gas expenditures, then you need to compute how much gas the contract can pay for. The *overhead* involved in execution is approximately 140,000 gas. The following formula should be a close approximation to how much gas a contract can afford.

- let `gasPrice` be the gas price for the executing transaction.
- let `balance` be the ether balance of the contract.
- let `claimerDeposit` be the claimer's deposit amount.
- let `basePayment` be the base payment amount for the contract. This may either be the value specified by the scheduler, or the `claimAmount` if the contract has been claimed.
- `gas = (balance - 2 * basePayment - claimerDeposit) / gasPrice`

CHAPTER 5

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.

These funds must be presented upfront at the time of scheduling and are held by the call contract until execution.

5.1 Call Payment and Fees

When a call is scheduled, the scheduler can either provide values for the payment and fee, or leave them off in favor of using the default values.

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

5.1.1 The GasPriceScalar multiplier

Both the payment and the fee are multiplied by the **GasPriceScalar**.

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.

This multiplier is computed with the following formula.

- *IF* `gasPrice > anchorGasPrice`

```
anchorGasPrice / gasPrice
```

- *IF* `gasPrice <= anchorGasPrice`

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

Where:

- **anchorGasPrice** 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 `anchorGasPrice` 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;

gasPrice	multiplier
15	1.20
16	1.17
17	1.13
18	1.09
19	1.05
20	1.00
21	0.95
22	0.91
23	0.87
24	0.83
25	0.80
26	0.77
27	0.74
28	0.71
29	0.69
30	0.67
31	0.65
32	0.63
33	0.61
34	0.59
35	0.57
36	0.56
37	0.54
38	0.53
39	0.51
40	0.50

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.

5.2 Overhead

The gas overhead that you can expect to pay for your function is about 130,000 gas.

CHAPTER 6

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.

6.1 Abstract Solidity Contracts

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

6.1.1 Abstract Scheduler Contract Source Code

The following abstract solidity contract can be used to interact with the scheduling contract from a solidity contract.

```
contract SchedulerAPI {
    /*
     *  Call Scheduling API
     */
    function getMinimumGracePeriod() constant returns (uint);
    function getDefaultPayment() constant returns (uint);
    function getDefaultFee() constant returns (uint);

    function isKnownCall(address callAddress) constant returns (bool);

    function scheduleCall(address contractAddress, bytes4 abiSignature, uint targetBlock) public retu
    function scheduleCall(address contractAddress, bytes4 abiSignature, uint targetBlock, uint sugges
    function scheduleCall(address contractAddress, bytes4 abiSignature, uint targetBlock, uint sugges
    function scheduleCall(address contractAddress, bytes4 abiSignature, uint targetBlock, uint sugges
    function scheduleCall(address contractAddress, bytes4 abiSignature, uint targetBlock, uint sugges

    /*
     *  Next Call API
     */
    function getCallWindowSize() constant returns (uint);
    function getNextCall(uint blockNumber) constant returns (bytes32);
    function getNextCallSibling(address callAddress) constant returns (bytes32);
}
```

6.1.2 Abstract Call Contract Source Code

The following abstract solidity contract can be used to interact with a call contract from a solidity contract.

```
contract CallContractAPI {
    bytes public callData;
    address public contractAddress;
    uint8 public gracePeriod;
    address public schedulerAddress;
    uint public suggestedGas;
    bool public isCancelled;
    bool public wasCalled;
    bool public wasSuccessful;
    uint public anchorGasPrice;
    uint public basePayment;
    bytes4 public abiSignature;
    uint public baseFee;
    uint public targetBlock;

    function execute() public;
    function cancel() public;

    function claim() public;

    address public claimer;
    uint public claimerDeposit;
    uint public claimAmount;

    function checkExecutionAuthorization(address executor, uint256 block_number) public returns (bool

    function getClaimAmountForBlock() public returns (uint);
    function getClaimAmountForBlock(uint256 block_number) public returns (uint);

    function registerData() public;
}
```

6.1.3 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 7

Events

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

7.1 Scheduler Events

The Scheduler contract logs the following events.

7.1.1 Call Scheduled

- **Solidity Event Signature:** `CallScheduled(address callAddress)`
- **ABI Signature:** `0x2b05d346`

Logged when a new scheduled call is created.

7.1.2 Call Rejected

- **Solidity Event Signature:** `CallRejected(address indexed schedulerAddress, bytes32 reason)`
- **ABI Signature:** `0x513485fc`

Logged when an attempt to schedule a function call fails.

7.2 Call Contract Events

Each CallContract logs the following events.

7.2.1 Call Executed

- **Solidity Event Signature:** `CallExecuted(address indexed executor, uint gasCost, uint payment, u` ;
- **ABI Signature:** `0x4538b7ec`

Executed when the call is executed.

7.2.2 Call Aborted

- **Solidity Event Signature:** `_CallAborted(address executor, bytes32 reason)`
- **ABI Signature:** `0xe92bb686`

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. (Note that this event name starts with an underscore)

CHAPTER 8

Terminology

Definitions for various terms that are used regularly to describe parts of the Alarm service.

8.1 General

Ethereum Alarm Clock, Alarm, Alarm Service Generic terms for the service as a whole.

Scheduler Contract The solidity contract responsible for scheduling a function call.

Call Contract The solidity contract that is deployed for each scheduled call. This contract handles execution of the call, registration of call data, gas reimbursment, and payment and fee dispursment.

Scheduled Call A contract function call that has been registered with the Alarm service to be executed at a specified time in the future (currently denoted by a block number).

8.2 Calls and Call Scheduling

Scheduler The account which scheduled the function call.

Executor The account which initiates the transaction which executes a scheduled function call.

Target Block The first block number that a scheduled call can be called.

Grace Period The number of blocks after the **target block** that a scheduled call can be be called.

Freeze Window The 10 blocks directly preceeding the target block for a call

Claim Window The 255 block window prior to the Freeze Window during which the call may be claimed for exclusive rights to execution during the first 16 blocks of the call window.

Call Window Used to refer to either the full window of blocks during which a scheduled call can be executed, or a portion of this window that has been designated to a specific caller.

Payment The ethereum amount that is paid to the executor of the scheduled call.

Fee The ethereum amount that is paid to the creator of the Alarm service.

Anchor Gas Price The gas price that was used when scheduling the scheduled call.

Gas Price Scalar A number ranging from 0 - 2 that is derived from the difference between the gas price of the executing transaction and the **anchor gas grice**. This number equals 1 when the two numbers are equal. It approaches 2 as the executing gas grice drops below the anchor gas price. It approaches zero as the executing gas price rises above the anchor gas price. This multiplier is applied to the payment and fee values, intending to motivate the executor to use a reasonable gas price.

CHAPTER 9

Changelog

9.1 0.5.0

- Each scheduled call now exists as it's own contract, referred to as a call contract.
- The authorization API has been removed. It is now possible for the contract being called to look up `msg.sender` on the scheduling contract and find out who scheduled the call.
- The account management API has been removed. Each call contract now manages it's own gas money, the remainder of which is given back to the scheduler after the call is executed.
- All of the information that used to be stored about the call execution is now placed in event logs (gasUsed, wasSuccessful, wasCalled, etc)

9.2 0.4.0

- Convert Alarm service to use library contracts for all functionality.
- CallerPool contract API is now integrated into the Alarm API

9.3 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.

9.4 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.

9.5 0.1.0

- Initial release.