
ENS Documentation

Release 0.1

Nick Johnson <nick@ethereum.org>

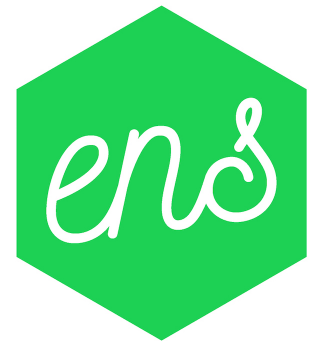
Sep 30, 2017

Contents:

1	Introduction	1
1.1	Overview	1
1.2	Architecture	2
1.3	Namehash	2
1.4	ENS on Ethereum	3
1.5	Resources	3
2	Quickstart	5
3	User Guide	7
3.1	Registering a name with the FIFS registrar	7
3.2	Registering a name with the auction registrar	8
3.3	Interacting with the ENS registry	10
3.4	Resolving ENS names	12
3.5	Reverse name resolution	13
4	Implementer's Guide	15
4.1	Writing a resolver	15
4.2	Resolving names onchain	16
4.3	Writing a registrar	17
4.4	Interacting with ENS offchain	17
4.5	Normalising and validating names	17
4.6	Handling of ambiguous names	17
5	Deploying ENS	19
5.1	Deploy the registry	19
5.2	Deploying a registrar	20
6	Hosting a DNS domain on ENS	23
7	FAQ	25
7.1	Why are names registered as hashes?	25
7.2	How do the DApp and the twitter bot know what names people are auctioning?	25
7.3	What is the disguise amount / extra amount?	25
7.4	Which wallets and DApps support ENS so far?	26
7.5	Why does it say my name isn't available yet?	26
7.6	How is the start time for each name determined?	26

7.7	Why is my reveal transaction failing?	26
7.8	I bid on an auction and didn't win; why don't I see the refund in my transaction list?	26
7.9	I bid on one name, but Etherscan shows me as bidding on other name(s)! Why?	27
7.10	Once I own a name, can I create my own subdomains?	27
7.11	Can I change the address my name points to after I've bought it?	27
7.12	How much do I have to deposit if I'm the only bidder?	27
7.13	Can I register a TLD of my own in the ENS?	27
7.14	Instead of burning funds in the auction for bidding costs and penalties, shouldn't they be donated to the Ethereum Foundation?	27
7.15	Who will own the ENS rootnode? What powers does that grant them?	27
7.16	What about foreign characters? What about upper case letters? Is any unicode character valid?	28
7.17	Nameprep isn't enforced in the ENS system, is this a security/spoofing/phishing concern?	28
7.18	How was the minimum character length of 7 chosen?	28
7.19	What values will the permanent registrar try to optimize for?	28
7.20	What kinds of behaviours are likely to result in losing ownership of a name?	29
7.21	Can the bids be revealed during the auction period?	29

8 Indices and tables 31



ENS is the Ethereum Name Service, a distributed, open, and extensible naming system based on the Ethereum blockchain.

ENS can be used to resolve a wide variety of resources. The initial standard for ENS defines resolution for Ethereum addresses, but the system is extensible by design, allowing more resource types to be resolved in future without the core components of ENS requiring upgrades.

Overview

The primary goal of ENS is to resolve human-readable names, like ‘myname.eth’, into machine-readable identifiers, including Ethereum addresses, Swarm and IPFS content hashes, and other identifiers. A secondary purpose is to provide metadata about names, such as ABIs for contracts, and whois information for users.

ENS has similar goals to DNS, the Internet’s Domain Name Service, but has significantly different architecture, due to the capabilities and constraints provided by the Ethereum blockchain. Like DNS, ENS operates on a system of dot-separated hierarchial names called domains, with the owner of a domain having full control over the distribution of subdomains.

Top-level domains, like ‘.eth’ and ‘.test’ are owned by smart contracts called registrars, which specify rules governing the allocation of their subdomains. Anyone may, by following the rules imposed by these registrar contracts, obtain

ownership of a second-level domain for their own use.

Architecture

ENS has two principal components: the registry, and resolvers.

The ENS registry consists of a single central contract that maintains a list of all domains and subdomains, and stores three critical pieces of information about each:

- The owner of the domain
- The resolver for the domain
- The time-to-live for all records under the domain

The owner of a domain may be either an external account (a user) or a smart contract. A registrar is simply a smart contract that owns a domain, and issues subdomains of that domain to users that follow some set of rules defined in the contract.

Owners of domains in the ENS registry may:

- Set the resolver and TTL for the domain
- Transfer ownership of the domain to another address
- Change the ownership of subdomains

The ENS registry is deliberately straightforward, and exists only to map from a name to the resolver responsible for it.

Resolvers are responsible for the actual process of translating names into addresses. Any contract that implements the relevant standards may act as a resolver in ENS. General-purpose resolver implementations are offered for users whose requirements are straightforward, such as serving an infrequently changed address for a name.

Each record type - Ethereum address, Swarm content hash, and so forth - defines a method or methods that a resolver must implement in order to provide records of that kind. New record types may be defined at any time via the EIP standardisation process, with no need to make changes to the ENS registry or to existing resolvers in order to support them.

Namehash

Names in ENS are represented as 32 byte hashes, rather than as plain text. This simplifies processing and storage, while permitting arbitrary length domain names, and preserves the privacy of names onchain. The algorithm used to translate domain names into hashes is called namehash. Namehash is defined in [EIP137](#).

In order to preserve the hierarchal nature of names, namehash is defined recursively, making it possible to derive the hash of a subdomain from the hash of the parent domain and the name or hash of the subdomain.

Terminology

- **domain** - the complete, human-readable form of a name; eg, *'vitalik.wallet.eth'*.
- **label** - a single component of a domain; eg, *'vitalik'*, *'wallet'*, or *'eth'*. A label may not contain a period ('.').
- **label hash** - the output of the keccak-256 function applied to a label; eg, *keccak256('eth') = 0x4f5b812789fc606be1b3b16908db13fc7a9adf7ca72641f84d75b47069d3d7f0*.
- **node** - the output of the namehash function, used to uniquely identify a name in ENS.

Algorithm

First, a domain is divided into labels by splitting on periods ('.'). So, 'vitalik.wallet.eth' becomes the list ['vitalik', 'wallet', 'eth'].

The namehash function is then defined recursively as follows:

```
namehash([]) = 0x0000000000000000000000000000000000000000000000000000000000000000
namehash([label, ...]) = keccak256(namehash(...), keccak256(label))
```

A sample implementation in Python is provided below.

```
def namehash(name):
    if name == '':
        return '\0' * 32
    else:
        label, _, remainder = name.partition('.')
        return sha3(namehash(remainder) + sha3(label))
```

ENS on Ethereum

ENS is deployed on mainnet at `0x314159265dd8dbb310642f98f50c066173c1259b`, where users may register names under the eth TLD, which uses an auction based registrar.

ENS is also deployed on the Ropsten testnet at `0x112234455c3a32fd11230c42e7bccd4a84e02010`. Users may register names under two top level domains:

- `.eth`, which uses an auction based registrar with the same functionality as the main network, and allows users to keep names indefinitely; see *Registering a name with the auction registrar*.
- `.test`, which allows anyone to claim an unused name for test purposes, which expires after 28 days; see *Registering a name with the FIFO registrar*.

ENS is also deployed on the Rinkeby testnet at `0xe7410170f87102df0055eb195163a03b7f2bff4a`, where only the `.test` top level domain is supported.

Resources

- [EIP137 - Ethereum Name Service](#)
- [EIP162 - Initial ENS Registrar Specification](#)
- [ethereum-ens Javascript library](#)
- Nick's talk on ENS at DevCon 2: <https://www.youtube.com/watch?v=pLDDbCZXvTE>
- DevCon 2 talk slides: <https://arachnid.github.io/devcon2/#/title>
- Web interface for Ethereum enabled browsers: <https://registrar.ens.domains>

CHAPTER 2

Quickstart

Just want to get a name and make it resolve to something? Here's how.

First, download `ensutils-testnet.js` to your local machine, and import it into an Ethereum console on a node synced to ropsten or rinkeby:

```
loadScript('/path/to/ensutils-testnet.js');
```

Before registering, check that nobody owns the name you want to register:

```
new Date(testRegistrar.expiryTimes(web3.sha3('myname')).toNumber() * 1000)
```

If this line returns a date earlier than the current date, the name is available and you're good to go. You can register the domain for yourself by running:

```
testRegistrar.register(web3.sha3('myname'), eth.accounts[0], {from: eth.accounts[0]})
```

Next, tell the ENS registry to use the public resolver for your name:

```
ens.setResolver(namehash('myname.test'), publicResolver.address, {from: eth.  
↪accounts[0]});
```

Once that transaction is mined, tell the resolver to resolve that name to your account:

```
publicResolver.setAddr(namehash('myname.test'), eth.accounts[0], {from: eth.  
↪accounts[0]});
```

...or any other address:

```
publicResolver.setAddr(namehash('myname.test'), '0x1234...', {from: eth.accounts[0]});
```

If you want, create a subdomain and do the whole thing all over again:

```
ens.setSubnodeOwner(namehash('myname.test'), web3.sha3('foo'), eth.accounts[1],
↳{from: eth.accounts[0]});
ens.setResolver(namehash('foo.myname.test'), publicResolver.address, {from: eth.
↳accounts[1]});
...
```

Finally, you can resolve your newly created name:

```
getAddr('myname.test')
```

which is shorthand for:

```
resolverContract.at(ens.resolver(namehash('myname.test'))).addr(namehash('myname.test
↳'))
```

This user guide is intended for anyone wanting to register, configure, and update ENS names using a Javascript console and web3.js. Before starting, open up a geth console, download `ensutils.js` or `ensutils-testnet.js` to your local machine, and import it into a running Ethereum console:

```
loadScript('/path/to/ensutils.js');
```

Registering a name with the FIFS registrar

The public ENS deployment on Ropsten uses a first-in-first served registrar for the `.test` top level domain. Domains on this TLD are configured to expire, allowing anyone else to claim them, 28 days after registration.

`ensutils.js` defines an object `testRegistrar`, for interacting with the registrar for the `.test` TLD. If you want to interact with a different first-in-first-served registrar, you can instantiate it with:

```
var myRegistrar = fifsRegistrarContract.at(address);
```

Before registering, check that nobody owns the name you want to register:

```
new Date(testRegistrar.expiryTimes(web3.sha3('myname')).toNumber() * 1000)
```

If this line returns a date earlier than the current date, the name is available and you're good to go.

The FIFS registrar's interface is extremely simple, and exposes a method called `register` that you can call with the (hashed) name you want to register and the address you want to own the name. To register a name, simply call that method to send a registration transaction:

```
testRegistrar.register(web3.sha3('myname'), eth.accounts[0], {from: eth.accounts[0]});
```

Once this transaction is mined, assuming the name was not already assigned, it will be assigned to you, and you can proceed to *Interacting with the ENS registry*.

Registering a name with the auction registrar

Once deployed on mainnet, ENS names will be handed out via an auction process, on the ‘.eth’ top-level domain. A preview of this is available on the Ropsten testnet, and you can register names via it right now. Any names you register will persist until launch on mainnet, at which point the auction registrar on Ropsten will be deprecated and eventually deleted.

This registrar implements a blind auction, and is described in [EIP162](#). Names are initially required to be at least 7 characters long.

Registering a name with the auction registrar is a multi-step process.

Starting an auction

Before placing a bid, you need to check if the name is available. Run this code to check:

```
ethRegistrar.entries(web3.sha3('name'))[0];
```

This will return a single integer between 0 and 5. The full solidity data structure for this can be viewed [here](#) in the Registrar contract. The numbers represent different ‘states’ a name is currently in.

- 0 - Name is available and the auction hasn’t started
- 1 - Name is available and the auction has been started
- 2 - Name is taken and currently owned by someone
- 3 - Name is forbidden
- 4 - Name is currently in the ‘reveal’ stage of the auction
- 5 - Name is not yet available due to the ‘soft launch’ of names.

If the returned value is 5, and is in the ‘soft launch’ is in effect; you can check when it will be available for auction with:

```
new Date(ethRegistrar.getAllowedTime(web3.sha3('name')) * 1000);
```

To start an auction for a name that’s not already up for auction, call *startAuction*:

```
ethRegistrar.startAuction(web3.sha3('name'), {from: eth.accounts[0], gas: 100000});
```

You can also start auctions for several names simultaneously, to disguise which name you’re actually interested in registering:

```
ethRegistrar.startAuctions([web3.sha3('decoy1'), web3.sha3('name'), web3.sha3('decoy2  
↪')], {from: eth.accounts[0], gas: 1000000});
```

Auctions normally run for 5 days: 3 days of bidding and 2 days of reveal phase. When initially deployed, there’s a “soft start” phase during which names are released for bidding gradually; this soft start lasts 4 weeks on ropsten, and 13 weeks on mainnet.

When a name is under auction, you can check the end time of the auction as follows:

```
new Date(ethRegistrar.entries(web3.sha3('name'))[2].toNumber() * 1000)
```

Placing a bid

Bids can be placed at any time during an auction except in the last 48 hours (the ‘reveal period’). Before trying to place a bid, make sure an auction is currently underway, as described above, and has more than 48 hours left to run.

To bid on an open auction, you need several pieces of data:

- The name you want to register
- The account you are sending the bid from
- The maximum amount you’re willing to pay for the name
- A random ‘salt’ value

In addition, you need to decide how much Ether you want to deposit with the bid. This must be at least as much as the value of your bid, but can be more, in order to disguise the true value of the bid.

First, start by generating a secret value. An easy way to do this is to use [random.org](https://www.random.org). Store this value somewhere secure - if you lose it, you lose your deposit, and your chance at winning the auction!

Now, you can generate your ‘sealed’ bid, with the following code:

```
var bid = ethRegistrar.shaBid(web3.sha3('name'), eth.accounts[0], web3.toWei(1, 'ether')
↳ ), web3.sha3('secret'));
```

The arguments are, in order, the name you want to register, the account you are sending the bid from, your maximum bid, and the secret value you generated earlier. Note that the bidding account will become the owner. You will lose funds if you seal with one account and send the bid with another!

Next, submit your bid to the registrar:

```
ethRegistrar.newBid(bid, {from: eth.accounts[0], value: web3.toWei(2, 'ether'), gas: 500000}
↳ );
```

In the example above, we’re sending 2 ether, even though our maximum bid is 1 ether; this is to disguise the true value of our bid. When we reveal our bid later, we will get the extra 1 ether back; the most we can pay for the name is 1 ether, as we specified when generating the bid.

Now it’s a matter of waiting until the reveal period before revealing your bid. Run the command to check the expiration date of the auction again, and make sure to come back in the final 48 hours of the auction:

```
new Date(ethRegistrar.entries(web3.sha3('name'))[2].toNumber() * 1000)
```

Revealing your bid

In order to win an auction, you must ‘reveal’ your bid. This is only possible during the ‘reveal’ phase, the last 48 hours of the auction, at which point new bids are prohibited. If you don’t reveal your bid by the time the auction ends, your deposit is forfeit - so make sure you store your salt in a safe place, and come back before the auction ends in order to reveal your bid.

To reveal, call the *unsealBid* function with the same values you provided earlier:

```
ethRegistrar.unsealBid(web3.sha3('name'), web3.toWei(1, 'ether'), web3.sha3('secret'),
↳ {from: eth.accounts[0], gas: 500000});
```

The arguments to *unsealBid* have the same order and meaning as those to *shaBid*, described in the bidding step, except that you don’t need to supply the account - it’s derived from your sending address.

After revealing your bid, the auction will be updated.

If your bid is less than a previously revealed bid, you will be refunded the whole amount of your bid.

If your bid is the largest revealed so far, you will be set as the current leading bidder. The difference between the actual amount of your bid and the amount you sent will be refunded immediately. The remainder - the actual bid - will stay locked. If you are later outbid it will be sent back to you.

Checking auctions

At any time, you can check the current winning bidder with:

```
deedContract.at(ethRegistrar.entries(web3.sha3('name'))[1]).owner();
```

and the value of the current winning bid with

```
web3.fromWei(ethRegistrar.entries(web3.sha3('name'))[4], 'ether');
```

Finalizing the auction

Once the auction has completed, it must be finalized in order for the name to be assigned to the winning bidder. Only the winning bidder can do this. To finalize, call the *finalizeAuction* function like so:

```
ethRegistrar.finalizeAuction(web3.sha3('name'), {from: eth.accounts[0], gas: 500000});
```

Once called, the winning bidder will be refunded the difference between their bid and the next highest bidder. If you're the only bidder, you get back all but 0.01 eth of your bid. The winner is then assigned the name in ENS.

If you are the winning bidder, congratulations!

Managing Ownership

After finalizing, your account now owns both the name in ENS and the deed in the Auction Registrar.

As the name owner, your account can manage the name using examples in “Interacting with the ENS registry”. For example, you can use `ens.setOwner` to transfer administration of the name to another account. The new name owner can manage that domain and all subdomains now. None of those actions affect your ownership of the deed.

As the deed owner, your account has the right to reset name ownership back to itself at any time, by using `ethRegistrar.finalizeAuction` again. You can also choose to transfer the deed to another account with:

```
ethRegistrar.transfer(web3.sha3('name'), newDeedOwnerAddress, {from: currentDeedOwnerAddress})
```

Caution: Transferring the deed is **irrevocable**. Be sure that you have verified the correct address for the new owner. Additionally, the ether you paid to win the auction will be transferred with the deed to the new owner.

Interacting with the ENS registry

The ENS registry forms the central component of ENS, mapping from hashed names to resolvers, as well as the owners of the names and their TTL (caching time-to-live).

Before you can make any changes to the ENS registry, you need to control an account that has ownership of a name in ENS. To obtain an ENS name on the Ropsten testnet, see [Registering a name with the auction registrar](#) for `.eth`, or [Registering a name with the FIFS registrar](#) for `.test`. Names on `.test` are temporary, and can be claimed by someone else 28 days later.

Alternately, you can obtain a subdomain from someone else who owns a domain, or [Deploying ENS](#). Note that while anyone can deploy their own ENS registry, those names will only be resolvable by users who reference that registry in their code.

Getting the owner of a name

You can retrieve the address of a name's owner using the *owner* function:

```
> ens.owner(namehash('somename.eth'));
"0xa303ddc620aa7d1390bacc8a495508b183fab59"
```

Getting the resolver for a name

You can retrieve the address of a name's resolver using the *resolver* function:

```
> ens.resolver(namehash('somename.eth'));
"0xc68de5b43c3d980b0c110a77a5f78d3c4c4d63b4"
```

Setting a name's resolver

You can set the resolver contract for a name using *setResolver*:

```
> ens.setResolver(namehash('somename.eth'), resolverAddress, {from: eth.accounts[0]});
```

A resolver is any contract that implements the resolver interface specified in [EIP137](#). You can deploy your own resolver, or you can use a publicly available one; on the mainnet, a simple resolver that supports `address` records and is usable by anyone is available; `ensutils.js` exposes it as *publicResolver*. To use it, first set it as the resolver for your name:

```
ens.setResolver(namehash('somename.eth'), publicResolver.address, {from: eth.
↪accounts[0]});
```

Then, call the resolver's *setAddr* method to set the address the name resolves to:

```
publicResolver.setAddr(namehash('somename.eth'), eth.accounts[0], {from: eth.
↪accounts[0]})
```

The above example configures `somename.eth` to resolve to the address of your primary account.

Transferring a name

You can transfer ownership of a name you own in the ENS registry to another account using *setOwner*:

```
> ens.setOwner(namehash('somename.eth'), newOwner, {from: eth.accounts[0]});
```

Note: If the name was acquired through a registrar, such as through a ‘.eth’ auction process, this will not transfer ownership of the locked bid. It will also not perform any administrative tasks that a registrar might want to do.

In general, to perform a complete transfer of a name acquired through a registrar, that particular registrar should be used as the interface. For the ‘.eth’ registrar, see [Managing Ownership](#).

Creating a subdomain

You can assign ownership of subdomains of any name you own with the `setSubnodeOwner` function. For instance, to create a subdomain ‘foo.somename.eth’ and set yourself as the owner:

```
> ens.setSubnodeOwner(namehash('somename.eth'), web3.sha3('foo'), eth.accounts[0],
↳ {from: eth.accounts[0]});
```

Or, to assign someone else as the owner:

```
> ens.setSubnodeOwner(namehash('somename.eth'), web3.sha3('foo'), someAccount, {from:
↳ eth.accounts[0]});
```

Note the use of `web3.sha3()` instead of `namehash()` when specifying the subdomain being allocated.

The owner of a name can reassign ownership of subdomains at any time, even if they’re owned by someone else.

Resolving Names

Now you’re ready to resolve your newly created name. For details how, read [Resolving ENS names](#).

Interacting with ENS from a DApp

An NPM module, [ethereum-ens](#), is available to facilitate interacting with the ENS from Javascript-based DApps.

Interacting with ENS from a contract

The [ENS registry interface](#) provides a Solidity definition of the methods available for interacting with the ENS. Using this, and the address of the ENS registry, contracts can read and write the ENS registry directly.

A Solidity library to facilitate this will be available soon.

Resolving ENS names

This page describes how ENS name resolution works at the contract level. For convenient use in DApps, an NPM package, [ethereum-ens](#) is available which abstracts away much of the detail and makes name resolution a straightforward process.

Step by step

Get the node ID (namehash output) for the name you want to resolve:

```
var node = namehash('myname.eth');
```

Ask the ENS registry for the resolver responsible for that node:

```
var resolverAddress = ens.resolver(node);
```

Create an instance of a resolver contract at that address:

```
var resolver = resolverContract.at(resolverAddress);
```

Finally, ask the resolver what the address is:

```
resolver.addr(node);
```

Oneliner

This statement is equivalent to all of the above:

```
resolverContract.at(ens.resolver(namehash('myname.eth'))).addr(namehash('myname.eth  
→'));
```

For convenience, `ensutils.js` provides a function, `getAddr` that does all of this for you with the default ENS registry:

```
getAddr('myname.eth')
```

Reverse name resolution

ENS also supports reverse resolution of Ethereum addresses. This allows an account (contract or external) to associate metadata with itself, such as its canonical name - 'Ethereum caller ID' if you will.

Reverse records are in the format `<ethereum address>.addr.reverse` - for instance, the official registry would have its reverse records at `314159265dd8dbb310642f98f50c066173c1259b.addr.reverse`.

`addr.reverse` has a registrar with `claim`, `claimWithResolver`, and `setName` functions.

The `claim` function takes one argument, the Ethereum address that should own the reverse record.

This permits a very simple pattern for contracts that wish to delegate control of their reverse record to their creator; they simply need to add this function call to their constructor:

```
reverseRegistrar.claim(msg.sender)
```

Claiming your account

Call the `claim` function on the `reverseRegistrar` object:

```
reverseRegistrar.claim(eth.accounts[0], {from: eth.accounts[0]});
```

After that transaction is mined, the appropriate reverse record is now owned by your account, and, you can deploy a resolver and set records on it; see [Interacting with the ENS registry](#) for details.

Alternately, you can claim and set the resolver record in one operation:

```
reverseRegistrar.claimWithResolver(eth.accounts[0], publicResolver.address, {from: eth.accounts[0]});
```

Setting up a reverse name for your address

If you just want to set up a reverse resolver with a name record, a quick convenience function is available in the reverse registrar:

```
reverseRegistrar.setName('myname.eth', {from: eth.accounts[0]});
```

This function points your reverse record at a default resolver, then sets the name record on that resolver for you - everything you need to set up 'caller ID' in a single transaction.

This section is intended to provide guidance for anyone wanting to implement tools and applications that use ENS, or custom resolvers within ENS.

Writing a resolver

Resolvers are specified in [EIP137](#). A resolver must implement the following method:

```
function supportsInterface(bytes4 interfaceID) constant returns (bool)
```

supportsInterface is defined in [EIP165](#), and allows callers to determine if a resolver supports a particular record type. Record types are specified as a set of one or more methods that a resolver must implement together. Currently defined record types include:

Record type	Function(s)	Interface ID	Defined in
Ethereum address	<i>addr</i>	0x3b3b57de	EIP137
ENS Name	<i>name</i>	0x691f3431	EIP181
ABI specification	<i>ABI</i>	0x2203ab56	EIP205
Public key	<i>pubkey</i>	0xc8690233	EIP619

supportsInterface must also return true for the *interfaceID* value *0x01ffc9a7*, which is the interface ID of *supportsInterface* itself.

Additionally, the *content()* interface is currently used as a defacto standard for Swarm hashes, pending standardisation, and has an interface ID of *0xd8389dc5*.

For example, a simple resolver that supports only the *addr* type might look something like this:

```
contract SimpleResolver {
    function supportsInterface(bytes4 interfaceID) constant returns (bool) {
        return interfaceID == 0x3b3b57de;
    }

    function addr(bytes32 nodeID) constant returns (address) {
```

```
    return address(this);
}
}
```

This trivial resolver always returns its own address as answer to all queries. Practical resolvers may use any mechanism they wish to determine what results to return, though they should be *constant*, and should minimise gas usage wherever possible.

NOTE: If you are resolving *addr()* records, you **MUST** treat a return value from the resolver of *0x00...00* as that record being unset. Failing to do so could result in users accidentally sending funds to the null address if they have configured a resolver in ENS, but not set the resolver record!

Resolving names onchain

Solidity libraries for onchain resolution are not yet available, but ENS resolution is straightforward enough it can be done trivially without a library. Contracts may use the following interfaces:

```
contract ENS {
    function owner(bytes32 node) constant returns (address);
    function resolver(bytes32 node) constant returns (Resolver);
    function ttl(bytes32 node) constant returns (uint64);
    function setOwner(bytes32 node, address owner);
    function setSubnodeOwner(bytes32 node, bytes32 label, address owner);
    function setResolver(bytes32 node, address resolver);
    function setTTL(bytes32 node, uint64 ttl);
}

contract Resolver {
    function addr(bytes32 node) constant returns (address);
}
```

For resolution, only the *resolver()* function in the ENS contract is required; other methods permit looking up owners, and updating ENS from within a contract that owns a name.

With these definitions, looking up a name given its node hash is straightforward:

```
contract MyContract {
    ENS ens;

    function MyContract(address ensAddress) {
        ens = ENS(ensAddress);
    }

    function resolve(bytes32 node) constant returns(address) {
        var resolver = ens.resolver(node)
        return resolver.addr(node);
    }
}
```

While it is possible for a contract to process a human-readable name into a node hash, we highly recommend working with node hashes instead, as they are easier to work with, and allow contracts to leave the complex work of normalising the name to their callers outside the blockchain. Where a contract always resolves the same names, those names may be converted to a node hash and stored in the contract as a constant.

Writing a registrar

A registrar in ENS is simply any contract that owns a name, and allocates subdomains of it according to some set of rules defined in the contract code. A trivial first in first served contract is demonstrated below, using the ENS interface definition defined earlier.

```

contract FIFSRegistrar {
    ENS ens;
    bytes32 rootNode;

    function FIFSRegistrar(address ensAddr, bytes32 node) {
        ens = ENS(ensAddr);
        rootNode = node;
    }

    function register(bytes32 subnode, address owner) {
        var node = sha3(rootNode, subnode);
        var currentOwner = ens.owner(node);

        if (currentOwner != 0 && currentOwner != msg.sender) throw;

        ens.setSubnodeOwner(rootNode, subnode, owner);
    }
}

```

Interacting with ENS offchain

A Javascript library, [ethereum-ens](#), is available to facilitate reading and writing ENS from offchain. This section will be updated as libraries for more languages become available.

Normalising and validating names

Before a name can be converted to a node hash using *Namehash*, the name must first be normalised and checked for validity - for instance, converting *fOO.eth* into *foo.eth*, and prohibiting names containing forbidden characters such as underscores. It is crucial that all applications follow the same set of rules for normalisation and validation, as otherwise two users entering the same name on different systems may resolve the same human-readable name into two different ENS names.

Applications using ENS and processing human-readable names must follow [UTS46](#) for normalisation and validation. Processing should be done with non-transitional rules, and with *UseSTD3ASCIIRules=true*.

The [ethereum-ens](#) Javascript library incorporates compliant preprocessing into its *validate* and *namehash* functions, so users of this library avoid the need to handle this manually.

Handling of ambiguous names

Because of the large number of characters in unicode, and the wide variety of scripts represented, inevitably there are different Unicode characters that are similar or even identical when shown in common fonts. This can be abused to trick users into thinking they are visiting one site or resource, when in fact they are visiting another. This is known as a [homoglyph attack](#).

User agents and other software that display names to users should take countermeasures against these attacks, such as by highlighting problematic characters, or showing warnings to users about mixed scripts. [Chromium's IDNA strategy](#) may serve as a useful reference for user-agent behaviour around rendering IDNA names.

Deploying ENS

If you'd like to deploy ENS on your own network, or deploy your own copy of ENS on a public network, this guide shows you how. If you want to use an existing ENS deployment, read *Interacting with the ENS registry* instead. If you want to register a name on the Ropsten (testnet) ENS deployment, read *Registering a name with the FIFS registrar* or *Registering a name with the auction registrar*.

Deploy the registry

First, you need to deploy ENS's central component, the registry. To do so, paste this code into an Ethereum console:

```

var ensContract = web3.eth.contract([{"constant":true,"inputs":[{"name":"node","type":
↪ "bytes32"}],"name":"resolver","outputs":[{"name":"address"}],"payable
↪ :false,"type":"function"}, {"constant":true,"inputs":[{"name":"node","type":"bytes32
↪ "}], "name":"owner","outputs":[{"name":"","type":"address"}],"payable":false,"type":
↪ "function"}, {"constant":false,"inputs":[{"name":"node","type":"bytes32"}, {"name":
↪ "label","type":"bytes32"}, {"name":"owner","type":"address"}],"name":"setSubnodeOwner
↪ ","outputs":[],"payable":false,"type":"function"}, {"constant":false,"inputs":[{"name
↪ ":"node","type":"bytes32"}, {"name":"ttl","type":"uint64"}],"name":"setTTL","outputs
↪ ": [],"payable":false,"type":"function"}, {"constant":true,"inputs":[{"name":"node",
↪ "type":"bytes32"}],"name":"ttl","outputs":[{"name":"","type":"uint64"}],"payable
↪ :false,"type":"function"}, {"constant":false,"inputs":[{"name":"node","type":
↪ "bytes32"}, {"name":"resolver","type":"address"}],"name":"setResolver","outputs":[],
↪ "payable":false,"type":"function"}, {"constant":false,"inputs":[{"name":"node","type
↪ ":"bytes32"}, {"name":"owner","type":"address"}],"name":"setOwner","outputs":[],
↪ "payable":false,"type":"function"}, {"anonymous":false,"inputs":[{"indexed":true,
↪ "name":"node","type":"bytes32"}, {"indexed":false,"name":"owner","type":"address"}],
↪ "name":"Transfer","type":"event"}, {"anonymous":false,"inputs":[{"indexed":true,"name
↪ ":"node","type":"bytes32"}, {"indexed":true,"name":"label","type":"bytes32"}, {
↪ "indexed":false,"name":"owner","type":"address"}],"name":"NewOwner","type":"event"},
↪ {"anonymous":false,"inputs":[{"indexed":true,"name":"node","type":"bytes32"}, {
↪ "indexed":false,"name":"resolver","type":"address"}],"name":"NewResolver","type":
↪ "event"}, {"anonymous":false,"inputs":[{"indexed":true,"name":"node","type":"bytes32
↪ "}], {"indexed":false,"name":"ttl","type":"uint64"}],"name":"NewTTL","type":"event"
↪ });
var ens = ensContract.new({
  from: web3.eth.accounts[0],
  data:
↪ "0x333600060000155610220806100146000396000f3630178b8bf60e060020a600035041415610023576020600435015460
↪ ",
  gas: 4700000
}, function (e, contract) {
  console.log(e, contract);
  if (typeof contract.address !== 'undefined') {
    console.log('Contract mined! address: ' + contract.address + '
↪ transactionHash: ' + contract.transactionHash);
  }
});

```

Once successfully mined, you will have a fresh ENS registry, whose root node is owned by the account that created the transaction (in this case, the first account on your node). This account has total control over the ENS registry - it can create and replace any node in the entire tree.

For instructions on how to interact with the registry, see *Interacting with the ENS registry*.

Deploying a registrar

A registrar is a contract that has ownership over a node (name) in the ENS registry, and provides an interface for users to register subnodes (subdomains). You can deploy a registrar on any name; in this example we'll deploy a simple first-in-first-served registrar for the root node.

To deploy a first-in-first-served registrar on the root node of an ENS registry you control, execute this code in an Ethereum console:

```

var registrarContract = web3.eth.contract([{"constant":false,"inputs":[{"name":
↪ "subnode","type":"bytes32"}, {"name":"owner","type":"address"}],"name":"register",
↪ "outputs":[],"payable":false,"type":"function"}, {"inputs":[{"name":"ensAddr","type":
↪ "address"}, {"name":"node","type":"bytes32"}, {"name":"_startDate","type":"uint256
↪ "}], "type":"constructor"}]);

```

Hosting a DNS domain on ENS

Experimental support is available for hosting DNS domains on the Ethereum blockchain via ENS. This works by configuring the domain's nameserver records to point to gateway DNS servers; these DNS servers resolve lookups by consulting an ENS registry which points to resolvers containing the zone data for the relevant domain.

The steps to host your own domain on the blockchain are:

1. deploy your own ENS registry
2. Deploy an instance of `DNSResolver`
3. Update your ENS registry to set your newly deployed `DNSResolver` as the resolver for your domain name (eg, by calling `setSubnodeOwner` and `setResolver`; see *Interacting with the ENS registry* for details). Don't forget to set the TTL on this record to a reasonable value, or your DNS data may not get cached.
4. Write a zonefile. The zonefile must include an NS record for your domain that specifies the resolver as `address.ns1.ens.domains`, where `address` is the address of the ENS registry you deployed in step 1, without the leading '0x'. An [example zonefile](#) is available for reference.
5. Clone and build `ensdns`. Start a local node, and run `'ensdns upload --keystore=path/to/keystore zonefile'` to upload the zone to the blockchain.
6. Update your NS records with your registrar to point to the name above (`address.ns1.ens.domains`).

Please note that this feature is still experimental, and shouldn't be used for anything production critical; the DNS gateway is lightly tested, and only a single instance is running at present, providing no backup or failover in case of server issues. The API and configuration may change in backwards-incompatible ways, breaking your nameserver!

Why are names registered as hashes?

Hashes are used for two main reasons. First, to prevent trivial enumeration of the entire set of domains, which helps preserve privacy of names (for instance, so you can register the domain for your startup before you publicly launch). Second, because hashes provide a fixed length identifier that can easily be passed around between contracts with fixed overhead and no issues around passing around variable-length strings.

How do the DApp and the twitter bot know what names people are auctioning?

The DApp and the twitter bot have built in lists of common names, drawn from an English dictionary and Alexa's list of top 1 million internet domain names. They use these lists to show you when common names are being auctioned. We do this because if the app didn't reveal these names, anyone with a little technical skill could find them out anyway, giving them an advantage over those who don't have the capacity to build their own list and code to check names against it.

What is the disguise amount / extra amount?

Although it's difficult for someone to determine what name you are bidding on, in some circumstances it's possible to either determine the exact name, or narrow it down to one of several. The disguise amount allows you to send extra ether along with your bid, to disguise the true amount of your bid. This ether is returned to you as soon as you reveal your bid.

Which wallets and DApps support ENS so far?

MyEtherWallet supports both registering names via the auction process and sending funds and interacting with contracts identified by their names.

Metamask supports sending funds to ENS names.

Mist is working on ENS support and should announce it soon.

LEth is working on ENS support and should announce it soon.

Status is working on ENS support and should announce it soon.

Why does it say my name isn't available yet?

ENS names are released gradually over a 'slow start' period of 8 weeks starting on May 4th 2017. The time at which any given name becomes available for auction during that period is effectively random. If you enter your desired name into the DApp, it will let you know when the first time you can auction it is.

How is the start time for each name determined?

Internally, we hash the name using keccak256, and express the result as a number between 0 and 1. Then, we multiply that by the duration of the launch period (8 weeks) and add that to the start date (May 4th 2017 1100 UTC) to generate the time at which that name can first be auctioned. You can see the code for this [here](#).

Why is my reveal transaction failing?

Confirm all the values you put in during the bid, and try using a blockchain explorer to confirm the state of the ENS auction. These are situations that cause errors during the reveal transaction:

- You never successfully bid
- You're trying to reveal too early
- The domain name is wrong
- The amount is wrong
- The secret phrase is wrong
- You already revealed

I bid on an auction and didn't win; why don't I see the refund in my transaction list?

Refunds are sent as part of reveal transactions - when you reveal, if someone already outbid you, or when you're outbid later, if not - and as part of finalise transactions. They're not separate transactions, so they don't show up as such. If you're using etherscan, check the 'internal transactions' tab to find your refund information.

I bid on one name, but Etherscan shows me as bidding on other name(s)! Why?

The registrar DApp opens up to three randomly selected auctions as ‘chaff’ every time you place a bid, to make it more difficult for people to guess what you are bidding on. The names you see are not necessarily the one you’re bidding on; that’s contained in the sealed bid hash that was submitted.

Once I own a name, can I create my own subdomains?

Yes! You can create whatever subdomains you wish, and assign ownership of them to other people if you desire. You can even set up your own registrar for your domain!

Can I change the address my name points to after I’ve bought it?

Yes, you can update the addresses and other resources pointed to by your name at any time.

How much do I have to deposit if I’m the only bidder?

If only one bid is revealed, that bidder deposits 0.01 ether, the minimum bid. The remainder of their bid is refunded as soon as they finalise the auction.

Can I register a TLD of my own in the ENS?

No, TLDs are restricted to only .eth (on mainnet), or .eth and .test (on Ropsten), plus any special purpose TLDs such as those required to permit reverse lookups. There are no immediate plans to invite proposals for additional TLDs. In large part this is to reduce the risk of a namespace collision with the IANA DNS namespace.

Instead of burning funds in the auction for bidding costs and penalties, shouldn’t they be donated to the Ethereum Foundation?

Burning is fairly rare in the current registrar; it only burns fees if you reveal an illegal bid, or fail to reveal a bid during the reveal period. In all other circumstances they’re refunded to users, either when you’re outbid or when you relinquish the name. A small portion (0.5%) of the bids are burned with the intent of creating a cost for a large amount of domains or for highly valuable domains without the intention of buying them.

Burning fees is impartial, and avoids both political concerns over the destination of the funds, and perverse incentives for the beneficiary of the fees. The value of the ether burned is not destroyed, but rather equally distributed among all ether holders.

Who will own the ENS rootnode? What powers does that grant them?

The root node will initially be owned by a multisig contract, with keys held by trustworthy individuals in the Ethereum community. The exact makeup of this has not yet been decided on. We expect that this will be very hands-off, with

the root ownership only used to effect administrative changes, such as the introduction of a new TLD, or to recover from an emergency such as a critical vulnerability in a TLD registrar.

In the long term, the plan is to define a governance process for operations on the root node, and transfer ownership to a contract that enforces this process.

Since the owner of a node can change ownership of any subnode, the owner of the root can change any node in the ENS tree.

What about foreign characters? What about upper case letters? Is any unicode character valid?

Since the ENS contracts only deal with hashes, they have no direct way to enforce limits on what can be registered; character length restrictions are implemented by allowing users to challenge a short name by providing its preimage to prove it's too short.

This means that you can in theory register both 'foo.eth' and 'FOO.eth', or even <picture of my cat>.eth. However, resolvers such as browsers and wallets should apply the nameprep algorithm to any names users enter before resolving; as a result, names that are not valid outputs of nameprep will not be resolvable by standard resolvers, making them effectively useless. DApps that assist users with registering names should prevent users from registering unresolvable names by using nameprep to preprocess names being requested for registration.

Nameprep isn't enforced in the ENS system, is this a security/spoofing/phishing concern?

It's not enforced by the ENS contracts, but as described, resolvers are expected to use it before resolving names. This means that non-nameprep names will not be resolvable.

How was the minimum character length of 7 chosen?

By an informal survey of common 'high value' short names. This restriction is intended to be lifted once the permanent registrar is in place.

What values will the permanent registrar try to optimize for?

This is something that the community will have to decide as part of the standardisation process for the permanent registrar. A few possible principles to consider include:

- Accessibility: Registering a new name should be as easy and straightforward as possible.
- Correct valuation: registering a known or popular name should be costly and intentional, not a matter of luck
- Fairness: The system should not unduly favor people who happen to be in the right place at the right time.
- Stability: Names should only be reallocated with the express will of the owner or according to objective rules that will be discussed and set with the whole community.
- Principle of least surprise: Wherever possible, names should resolve to the resource most users would expect it to resolve to.

What kinds of behaviours are likely to result in losing ownership of a name?

This is the most important aspect to be decided on the Permanent registrar and the one we want more open debate. At minimum we want the owner of a name to have to execute some periodical transaction, just to prove that name hasn't been abandoned or keys have been lost. This transaction would probably also require additional ether to be locked or burned. The method to which that amount is calculated is yet to be determined but would probably be dependent on some (but not necessarily all) of these factors:

- The amount of ethers the domain was bought for originally
- The average cost of a domain back when it was first bought
- The average cost of a domain at the moment of renewal
- The current market value of the domain (to be calculated with some auction method)
- Other factors to be discussed

Just like the current model, this “fee” would not go to the Ethereum Foundation or any third party, but be locked or burned. Ideally, this financial (opportunity and liquidity) cost will make name squatting unprofitable – or at least make the name reselling market a dynamic and competitive one, focused on quick turnout and not on holding names long term for as much money as possible.

Another very possible option creating some sort of dispute resolution process for names, to ensure the “principle of least surprise” but this is a controversial idea and there are no clear ideas on how this process could be achieved in a fair way without risks of centralization and abuse of power.

Can the bids be revealed during the auction period?

No, you can't reveal during the auction period, only during the reveal period. Bids revealed during the auction period will have no effect and wouldn't count for the reveal period.

This document is licensed under the `@emph{Creative Commons Attribution License}`. To view a copy of this license, visit <http://creativecommons.org/licenses/by/2.0/>

CHAPTER 8

Indices and tables

- `genindex`
- `modindex`
- `search`