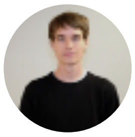

Peter Watts Follow
Platform Lead @ PROPSproject.com
Feb 15 · 6 min read

Beyond ERC20 — A guide to non-standard Ethereum token functionality

Ethereum ERC20 tokens are often talked about as a single category, but if you dig into the source code of the most popular tokens, you'll actually find that there's an incredible amount of diversity between them. The ERC20 standard merely outlines a *minimum* set of functionality that a token must adhere to. Most tokens go beyond this, adding all sorts of features. This post attempts to document this landscape of "non-standard" functionality, to show what's already being used in the wild, and give a glimpse of what the future holds.

Note: the focus is on classic, fungible tokens, rather than non-fungible ones.

ERC20

Before jumping into the non-standard, let's take a quick look at the standard. A token that adheres to the ERC20 spec implements the following functions:

totalSupply
Get the total token supply

balanceOf
Get the token balance of an address

transfer
Send tokens to another address

approve
Give another address permission to spend a certain allowance of tokens

allowance
Check the allowance that one address has for another

transferFrom
Spend the allowance that one address has for another

That's it. These are the only *required* functions.

Common Additions

This next set of features may not be standardized, but nearly every token includes one or all of them. Where possible, I've included links to the source code of at least one major token implementing the feature, so you can see how it works.

Minting / Crowd Sales
Most tokens need a way to establish the initial allocations. Typically, this either done by building distribution / crowdsale logic directly into the token contract [SNT, RDN, REQ], or by including a generic `mint()` function that can be called by an external contract [OMG].

Pausing
In many cases, a developer doesn't want their token to be transferable during some initial period (e.g. while a crowdsale is taking place), so they include the ability to pause/unpause transfers [MANA].

Ownership
It would be problematic if any user could arbitrarily mint or pause the token, so an ownership model is usually included, that limits certain actions to the owner [KNC].

Burning
Many tokens include some form of burning in their economic model. Rather than do it unofficially, they introduce a `burn()` function, that keeps the token's "totalSupply" value up to date [QSP].

Unlimited Allowance

A common pattern when using `approve()` is to grant a very large allowance to a trusted contract (e.g. this is how the 0x protocol works). To make this simpler, some tokens consider an allowance equal to `MAX_UINT` to be unlimited [ZRX].

Increase / Decrease Approval

One problem with the `approve()` function is that it's vulnerable to a front-running attack when trying to update an existing allowance. To avoid this, newer tokens use `increaseApproval()` and `decreaseApproval()` [LINK].

Advanced Features

Now we're getting to the fun stuff. These features are rarer, but can add significant flexibility to a token.

Vesting / Time locking
Often the creators opt to have their tokens subject to a vesting schedule [ANT] or time lock [OMG] that prevents them from immediately transferring them. Vesting can either be built directly into the token, or implemented with a dedicated contract that holds the locked tokens.

Upgrading / Migrating
If something were to go wrong, or a new token standard was to emerge (see below), a token issuer may wish to upgrade [PPT] or migrate [GNT] the token to a new address.

Cloning
This one is weird / cool. Tokens that are built off the MiniMe library [SNT, ANT, SWT] have the ability to be cloned, with the new token taking on a distribution equal to the original token's distribution at any previous block.

Restricted Ownership
Securities tokens are becoming increasingly popular, however they come with the limitation that only accredited investors can hold them. To

support this, protocols like R-Token extend ERC20 to only allow transfers from addresses that have been whitelisted in a registry.

Liquidity
The Bancor token [BNT] implements a continuous supply model that allows you to buy and sell tokens directly from the smart contract, which holds Ether in reserve, and dynamically sets a price based on supply and demand.

Delegated Transfers
Holding ETH in order to pay gas fees on token transfers can be inconvenient for end-users. As a workaround, tokens can allow transfers to be submitted by a 3rd party, who pays the gas and collects a token fee off the user [PROPS].

Potential Future Standards

Many of the features mentioned above don't need to become standards. It is perfectly OK to implement them ad-hoc, depending on the needs of the token. But other features, particularly those that establish new patterns for how tokens and contracts should interact, would benefit from becoming standardized.

Transfer And Call
If you want to send ERC20 tokens to a smart contract, and have it do something with those tokens, it currently takes two transactions. First the user "approves" an allowance for the contract, and then the contract utilizes the tokens via the "transferFrom" function.

To get around this, some tokens implemented "approveAndCall", which triggers a "receiveApproval" function in the receiving contract, allowing it to use the tokens in the same transaction. However, it never really took off as a widely adopted standard, and as a result, not many contracts handle it.

Since then, there have been many more attempts to add this elusive functionality, including ERC223, ERC677, ERC777 and ERC827, but none have managed to reach consensus, or see widespread deployment.

Preventing Token Loss

When you send Ether to a contract that can't handle it, the transaction fails. But if you do the same with tokens, they get trapped in the contract forever. This has led to millions of dollars worth of tokens getting lost. You can see for yourself by visiting the Etherscan page of any token, and looking at it's token balances (e.g. EOS has $800,000 trapped).

Many newer token contracts add functions that allow the owner to recover any tokens that are accidentally sent to it [FUN, BNT, KNC]. However, this is just a patch, and doesn't prevent tokens being sent to other contracts. ERC223 and ERC777 (discussed in more depth below) attempt to solve this by requiring contracts to explicitly indicate that they accept tokens, and aborting the transfer if they don't.

However, there is still ongoing debate over exactly how and when these scenarios should be handled, so nothing has been standardized yet.

Tokens 2.0

Amongst all the debate, there are two clear contenders with the potential to become the next standard in Ethereum tokens.

ERC223

Initially proposed almost a year ago, this standard aimed to address the two major issues mentioned above (transfer and call + lost tokens). However, the key supporters do not see backwards compatibility with ERC20 tokens and contracts as a high priority, instead emphasizing the need to move on:

> *The point of a new standard is to **NOT** be compatible with ERC20 because ERC20 has **design flaws**.*

This philosophy has caused a lot of disagreement (which is ongoing to this day), and as a result, adoption has been slow.

ERC777

In response to the stagnation of ERC223, ERC777 was proposed and has quickly gathered support from the community. In addition to addressing the key shortcomings of ERC20 (while maintaining full backwards compatibility), it also incorporates some of the learnings around how tokens are being used in the wild. This includes creating a standard for common actions like minting/burning, and formalizing the concept of “authorized operators” who can spend on your behalf.

For a more in-depth look at ERC777, check out this summary reddit post, or read the official issue on Github.

While it has promising momentum, it’s important to note that it is still being finalized, and there are no major tokens implementing it yet (Aragon platform tokens may be one of the first).

Takeaways

If you’re implementing a token, here are some things to keep in mind:

- ERC20 is still king. No other standard has settled yet.
- If you want to influence future standards, get involved with ERC777
- Don’t be afraid to include non-standard functionality if you think it will benefit your users. Just make sure it’s audited!

Speaking of which, if you were clicking the links through to source code, you may have noticed a lot of similar looking functions between tokens. That’s because nearly all of them are based off one of these battle-tested libraries:

- OpenZeppelin [OMG]
- MiniMe Token [SNT]
- Consensys [QTUM]
- DS Token [EOS]

Not only can you get an ERC20 template from these libraries, OpenZeppelin also offers ready made implementations of many non-standard features too.

Missing something?

Everything discussed above is probably just scratching the surface of Ethereum token functionality that’s out there in the wild. If you know of something else interesting, leave a comment or send me a tweet @ptrwtts