Units and Globally Available Variables

Ether Units

A literal number can take a suffix of `wei`, `finney`, `szabo` or `ether` to convert between the subdenominations of Ether, where Ether currency numbers without a postfix are assumed to be Wei, e.g. `2 ether == 2000 finney` evaluates to `true`.

Time Units

Suffixes like `seconds`, `minutes`, `hours`, `days`, `weeks` and `years` after literal numbers can be used to convert between units of time where seconds are the base unit and units are considered naively in the following way:

- `1 == 1 seconds`
- `1 minutes == 60 seconds`
- `1 hours == 60 minutes`
- `1 days == 24 hours`
- `1 weeks == 7 days`
- `1 years == 365 days`

Take care if you perform calendar calculations using these units, because not every year equals 365 days and not even every day has 24 hours because of leap seconds. Due to the fact that leap seconds cannot be predicted, an exact calendar library has to be updated by an external oracle.

Note

The suffix `years` has been deprecated due to the reasons above.

These suffixes cannot be applied to variables. If you want to interpret some input variable in e.g. days, you can do it in the following way:

```
function f(uint start, uint daysAfter) public {
    if (now >= start + daysAfter * 1 days) {
      // ...
    }
}
```

Special Variables and Functions

There are special variables and functions which always exist in the global namespace and are mainly used to provide information about the blockchain or are general-use utility functions.

Block and Transaction Properties

- `block.blockhash(uint blockNumber) returns (bytes32)`: hash of the given block - only works for 256 most recent, excluding current, blocks - deprecated in version 0.4.22 and replaced by `blockhash(uint blockNumber)`.
- `block.coinbase` (`address`): current block miner's address
- `block.difficulty` (`uint`): current block difficulty
- `block.gaslimit` (`uint`): current block gaslimit
- `block.number` (`uint`): current block number
- `block.timestamp` (`uint`): current block timestamp as seconds since unix epoch
- `gasleft() returns (uint256)`: remaining gas
- `msg.data` (`bytes`): complete calldata
- `msg.gas` (`uint`): remaining gas - deprecated in version 0.4.21 and to be replaced by `gasleft()`
- `msg.sender` (`address`): sender of the message (current call)
- `msg.sig` (`bytes4`): first four bytes of the calldata (i.e. function identifier)
- `msg.value` (`uint`): number of wei sent with the message
- `now` (`uint`): current block timestamp (alias for `block.timestamp`)
- `tx.gasprice` (`uint`): gas price of the transaction
- `tx.origin` (`address`): sender of the transaction (full call chain)

Note

The values of all members of `msg`, including `msg.sender` and `msg.value` can change for every **external** function call. This includes calls to library functions.

Note

Do not rely on `block.timestamp`, `now` and `blockhash` as a source of randomness, unless you know what you are doing.

Both the timestamp and the block hash can be influenced by miners to some degree. Bad actors in the mining community can for example run a casino payout function on a chosen hash and just retry a different hash if they did not receive any money.

The current block timestamp must be strictly larger than the timestamp of the last block, but the only guarantee is that it will be somewhere between the timestamps of two consecutive blocks in the canonical chain.

Note

The block hashes are not available for all blocks for scalability reasons. You can only access the hashes of the most recent 256 blocks, all other values will be zero.

ABI Encoding Functions

- `abi.encode(...) returns (bytes)`: ABI-encodes the given arguments
- `abi.encodePacked(...) returns (bytes)`: Performes packed encoding of the given arguments
- **`abi.encodeWithSelector(bytes4 selector, ...) returns (bytes)`: ABI-encodes the given arguments**

  starting from the second and prepends the given four-byte selector

- `abi.encodeWithSignature(string signature, ...) returns (bytes)`: Equivalent to `abi.encodeWithSelector(bytes4(keccak256(signature), ...)``

Note

These encoding functions can be used to craft data for function calls without actually calling a function. Furthermore, `keccak256(abi.encodePacked(a, b))` is a more explicit way to compute `keccak256(a, b)`, which will be deprecated in future versions.

See the documentation about the ABI and the tightly packed encoding for details about the encoding.

Error Handling

`assert(bool condition)`:
invalidates the transaction if the condition is not met - to be used for internal errors.

`require(bool condition)`:
reverts if the condition is not met - to be used for errors in inputs or external components.

`require(bool condition, string message)`:
reverts if the condition is not met - to be used for errors in inputs or external components. Also provides an error message.

`revert()`:
abort execution and revert state changes

`revert(string reason)`:
abort execution and revert state changes, providing an explanatory string

Mathematical and Cryptographic Functions

`addmod(uint x, uint y, uint k) returns (uint)`:
: compute `(x + y) % k` where the addition is performed with arbitrary precision and does not wrap around at `2**256`. Assert that `k != 0` starting from version 0.5.0.

`mulmod(uint x, uint y, uint k) returns (uint)`:
: compute `(x * y) % k` where the multiplication is performed with arbitrary precision and does not wrap around at `2**256`. Assert that `k != 0` starting from version 0.5.0.

`keccak256(...) returns (bytes32)`:
: compute the Ethereum-SHA-3 (Keccak-256) hash of the (tightly packed) arguments

`sha256(...) returns (bytes32)`:
: compute the SHA-256 hash of the (tightly packed) arguments

`sha3(...) returns (bytes32)`:
: alias to `keccak256`

`ripemd160(...) returns (bytes20)`:
: compute RIPEMD-160 hash of the (tightly packed) arguments

`ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (address)`:
: recover the address associated with the public key from elliptic curve signature or return zero on error (example usage)

In the above, “tightly packed” means that the arguments are concatenated without padding. This means that the following are all identical:

```
keccak256("ab", "c")
keccak256("abc")
keccak256(0x616263)
keccak256(6382179)
keccak256(97, 98, 99)
```

If padding is needed, explicit type conversions can be used: `keccak256("\x00\x12")` is the same as `keccak256(uint16(0x12))`.

Note that constants will be packed using the minimum number of bytes required to store them. This means that, for example, `keccak256(0) == keccak256(uint8(0))` and `keccak256(0x12345678) == keccak256(uint32(0x12345678))`.

It might be that you run into Out-of-Gas for `sha256`, `ripemd160` or `ecrecover` on a *private blockchain*. The reason for this is that those are implemented as so-called precompiled contracts and these contracts only really exist after they received the first message (although their contract code is hardcoded). Messages to non-existing contracts are more expensive and thus the execution runs into an Out-of-Gas error. A workaround for this problem is to first send e.g. 1 Wei to each of the contracts before you use them in your actual contracts. This is not an issue on the official or test net.

Address Related

`<address>.balance` (`uint256`):
: balance of the Address in Wei

`<address>.transfer(uint256 amount)`:
: send given amount of Wei to Address, throws on failure, forwards 2300 gas stipend, not adjustable

`<address>.send(uint256 amount) returns (bool)`:
: send given amount of Wei to Address, returns `false` on failure, forwards 2300 gas stipend, not adjustable

`<address>.call(...) returns (bool)`:
: issue low-level `CALL`, returns `false` on failure, forwards all available gas, adjustable

`<address>.callcode(...) returns (bool)`:
: issue low-level `CALLCODE`, returns `false` on failure, forwards all available gas, adjustable

`<address>.delegatecall(...) returns (bool)`:
: issue low-level `DELEGATECALL`, returns `false` on failure, forwards all available gas, adjustable

For more information, see the section on Address.

Warning

There are some dangers in using `send`: The transfer fails if the call stack depth is at 1024 (this can always be forced by the caller) and it also fails if the recipient runs out of gas. So in order to make safe Ether transfers, always check the return value of `send`, use `transfer` or even better: Use a pattern where the recipient withdraws the money.

Note

If storage variables are accessed via a low-level delegatecall, the storage layout of the two contracts must align in order for the called contract to correctly access the storage variables of the calling contract by name. This is of course not the case if storage pointers are passed as function arguments as in the case for the high-level libraries.

Note

The use of `callcode` is discouraged and will be removed in the future.

Contract Related

`this` **(current contract's type):**
: the current contract, explicitly convertible to Address

`selfdestruct(address recipient)`:
: destroy the current contract, sending its funds to the given Address

`suicide(address recipient)`:
: deprecated alias to `selfdestruct`

Furthermore, all functions of the current contract are callable directly including the current function.