# Solidity Inline-Assembly for Developer Experience

vectorized.eth

The beginner uses assembly to save gas.

The intermediate uses assembly to avoid the spurious dragon limit.

The advanced uses assembly to save time.

# Examples

- SafeTransferLib
- LibClone (ERC-7760 minimal upgradeable proxies)
- LibSort
- TestPlus

# SafeTransferLib

```
/// @dev Force sends `amount` (in wei) ETH to `to`, with a `gasStipend`.
function forceSafeTransferETH(address to, uint256 amount, uint256 gasStipend) internal {
    /// @solidity memory-safe-assembly
    assembly {
        if lt(selfbalance(), amount) {
            mstore(0x00, 0xb12d13eb) // `ETHTransferFailed()`.
            revert(0x1c, 0x04)
        }
        if iszero(call(gasStipend, to, amount, codesize(), 0x00, codesize(), 0x00)) {
            mstore(0x00, to) // Store the address in scratch space.
            mstore8(0x0b, 0x73) // Opcode `PUSH20`.
            mstore8(0x20, 0xff) // Opcode `SELFDESTRUCT`.
            if iszero(create(amount, 0x0b, 0x16)) { revert(codesize(), codesize()) } // For gas estimation.
        }
    }
}
```

The forceSafeTransferETH functions prevents gas-griefing.
Useful for auctions where the next bidder refunds the previous bidder.

WETH does not have the same address across all chains.

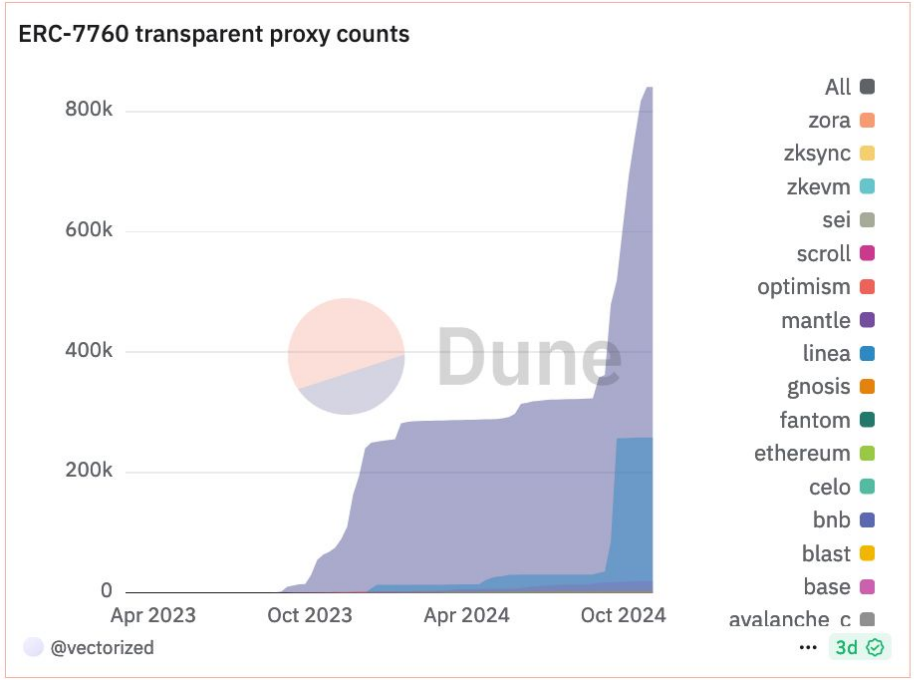
# LibClone

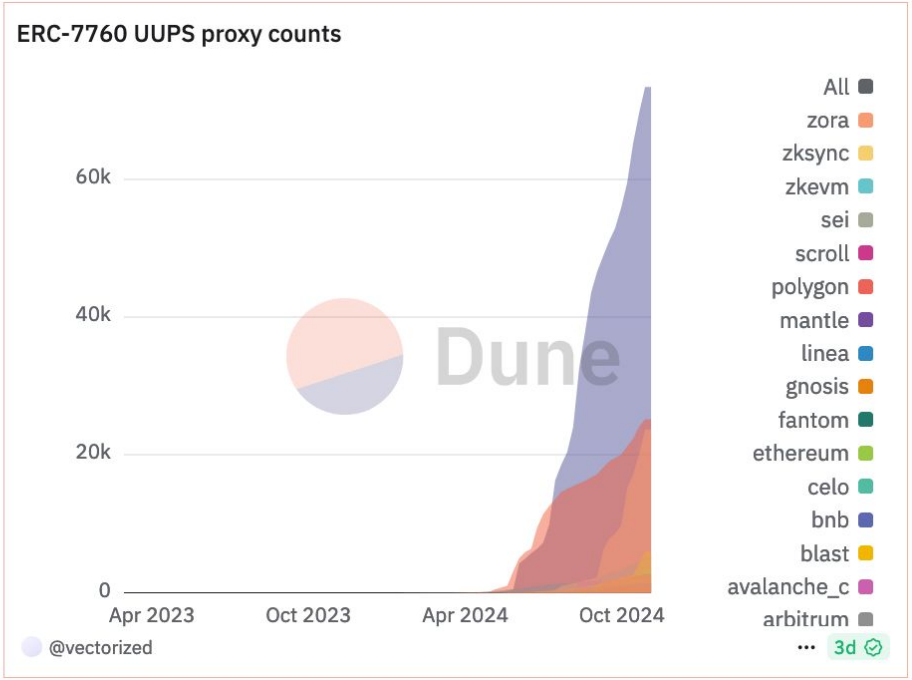
```
/// @dev Deploys a minimal ERC1967 proxy with `implementation` and `args`.
/// Deposits `value` ETH during deployment.
function deployERC1967(uint256 value, address implementation, bytes memory args)
    internal
    returns (address instance)
{
    /// @solidity memory-safe-assembly
    assembly {
        let m := mload(0x40)
        let n := mload(args)
        pop(staticcall(gas(), 4, add(args, 0x20), n, add(m, 0x60), n))
        mstore(add(m, 0x40), 0xcc3735a920a3ca505d382bbc545af43d6000803e6038573d6000fd5b3d6000f3)
        mstore(add(m, 0x20), 0x5155f3363d3d373d3d363d7f360894a13ba1a3210667c828492db98dca3e2076)
        mstore(0x16, 0x6009)
        mstore(0x14, implementation)
        // Do a out-of-gas revert if `n` is greater than `0xffff - 0x3d = 0xffc2`.
        mstore(gt(n, 0xffc2), add(0xfe61003d3d8160233d3973, shl(56, n)))
        mstore(m, mload(0x16))
        instance := create(value, m, add(n, 0x60))
        if iszero(instance) {
            mstore(0x00, 0x30116425) // `DeploymentFailed()`.
            revert(0x1c, 0x04)
        }
    }
}
```

The minimal upgradeable proxies (ERC-7760) in LibClone are auto-verified on Etherscan.

We've provide initcode functions in LibClone, you can deploy with any available factory (e.g. Nick's, CreateX).

# ERC-7760 Adoption



# LibSort

```
LibSort.sort(a);
LibSort.sort(b);
LibSort.uniquifySorted(a);
LibSort.uniquifySorted(b);
(bool found, uint256 index) = LibSort.searchSorted(a, needle); // `O(log n)`;
bool contains = LibSort.inSorted(a, needle); // `O(log n)`;
uint256[] memory difference = LibSort.difference(a, b);
uint256[] memory intersection = LibSort.intersection(a, b);
uint256[] memory union = LibSort.union(a, b);
```

Very useful for tests.

# TestPlus

```
function testHeapPushAndPop(uint256) public {
    unchecked {
        uint256 n = _random() % 8;
        uint256[] memory a = new uint256[](n);

        for (uint256 i; i < n; ++i) {
            uint256 r = _random();
            a[i] = r;
            heap0.push(r);
        }
        LibSort.insertionSort(a);
        for (uint256 i; i < n; ++i) {
            assertEq(heap0.pop(), a[i]);
        }
        assertEq(heap0.length(), 0);
    }
}
```

Solady's TestPlus allows you to use _random() to get a random number anywhere in your tests.

_random() uses biased sampling to allow the tests to hit edge cases.

We use inline-assembly to speed up test compilation and fuzzing.

We have scripts to help you install Solady's TestPlus easily in any Foundry project: https://github.com/Vectorized/foundry-starter

# Other Libraries You Should Check Out

- DynamicBufferLib
- DynamicArrayLib
- LibString
- SignatureCheckerLib
- SSTORE2
- LibZip
- CREATE3
- EfficientHashLib
- FixedPointMathLib

# Techniques

- Avoiding stack-too-deep
- Cool math
- Compile time zero-cost abstraction

# Avoiding stack-too-deep

Solady is tested with and without `--via-ir` from Solidity `^0.8.4`.

We insist on an *it-just-works* experience.

```
// Evaluate using a (8, 8)-term rational approximation.
// `p` is made monic, we will multiply by a scale factor later.
// forgefmt: disable-next-item
let p := sub( // This heavily nested expression is to avoid stack-too-deep for via-ir.
    sar(96, mul(add(43456485725739037958740375743393,
    sar(96, mul(add(24828157081833163892658089445524,
    sar(96, mul(add(32732854596385238486322540662 96,
        x), x))), x))), x)), 11111509109440967052023855526967)
p := sub(sar(96, mul(p, x)), 45023709667254063763336534515857)
p := sub(sar(96, mul(p, x)), 14706773417378608786704636184526)
p := sub(mul(p, x), shl(96, 795164235651350426258249787498))
// We leave `p` in `2**192` basis so we don't need to scale it back up for the division.
```

This snippet is from `FixedPointMathLib.lnWad`.

# Cool Math

```
/// @dev Returns the log2 of `x`.
/// Equivalent to computing the index of the most significant bit (MSB) of `x`.
/// Returns 0 if `x` is zero.
function log2(uint256 x) internal pure returns (uint256 r) {
    /// @solidity memory-safe-assembly
    assembly {
        r := shl(7, lt(0xffffffffffffffffffffffffffffffff, x))
        r := or(r, shl(6, lt(0xffffffffffffffff, shr(r, x))))
        r := or(r, shl(5, lt(0xffffffff, shr(r, x))))
        r := or(r, shl(4, lt(0xffff, shr(r, x))))
        r := or(r, shl(3, lt(0xff, shr(r, x))))
        // forgefmt: disable-next-item
        r := or(r, byte(and(0x1f, shr(shr(r, x), 0x8421084210842108cc6318c6db6d54be)),
            0x0706060506020504060203020504030106050205030304010505030400000000))
    }
}
```

Fully branchless log2.

Branchless functions can be easily inlined by Solidity.

Once inlined, values can be computed on compile-time if possible.

# Zero-cost Abstractions

```
/// @dev Returns whether `a` equals `b`, where `b` is a null-terminated small string.
function eqs(string memory a, bytes32 b) internal pure returns (bool result) {
    /// @solidity memory-safe-assembly
    assembly {
        // These should be evaluated on compile time, as far as possible.
        let m := not(shl(7, div(not(iszero(b)), 255))) // `0x7f7f ...`.
        let x := not(or(m, or(b, add(m, and(b, m)))))
        let r := shl(7, iszero(iszero(shr(128, x))))
        r := or(r, shl(6, iszero(iszero(shr(64, shr(r, x))))))
        r := or(r, shl(5, lt(0xffffffff, shr(r, x))))
        r := or(r, shl(4, lt(0xffff, shr(r, x))))
        r := or(r, shl(3, lt(0xff, shr(r, x))))
        // forgefmt: disable-next-item
        result := gt(eq(mload(a), add(iszero(x), xor(31, shr(3, r)))),
            xor(shr(add(8, r), b), shr(add(8, r), mload(add(a, 0x20)))))
    }
}
```

`LibString.eqs(a, "hehe")`

compiles into

```
and(eq(mload(a), 4),
eq(mload(add(a, 0x20)), 0x68656865))
```

There’s no memory allocation.

Inlinable branchless code enables zero-cost abstractions.

# Encapsulation

Solady encapsulates all the low-level logic away in elegant APIs.

With Solady, you can write clean, performant, high-level Solidity without touching inline-assembly.

# Polaris

- Make Solidity the best smart contract language.
  - If it is a useful feature to have onchain, that can be feasibly implemented within the limits of the Turing Completeness of the EVM, we can implement it (e.g. `FixedPointMathLib.lambertW0Wad`).
- Scale Ethereum through sheer app layer optimization.
- Make beautiful code that is art to be used by billions.

# Future Plans

Solady EOF ;)

https://github.com/Vectorized/solady/issues/1142

# Links

Solady: solady.org

Github: github.com/Vectorized

X: x.com/optimizoor <- For bite-sized Solidity lessons.