# Uniswap v4: Technical Report

**Protocol:** Uniswap v4

**Solidity Version:** `^0.8.24`

**Scope:** v4-core (`PoolManager`, `Pool`, `Hooks`, `Position`, `Tick`, `TickBitmap`) + v4-periphery (`PositionManager`, `V4Router`, `StateLibrary`)

**Analyzed by:** rubencrxz

## Table of Contents



## 1. Protocol Overview

Uniswap v4 is a complete architectural redesign of the AMM primitive, introducing three foundational innovations that compound together:

1. **Singleton PoolManager**: All liquidity pools live inside a single contract. There are no longer individual pair/pool contracts per token pair. This collapses multi-hop swap gas costs, eliminates approve-per-pool overhead, and centralizes token accounting.

2. **Hooks**: Arbitrary smart contract logic can be attached to every lifecycle event of a pool (swap, liquidity change, donate, initialize). Hooks enable on-chain limit orders, dynamic fee models, TWAMM integrations, yield-bearing LPs, MEV capture, and more — all without forking the core protocol.

3. **Flash Accounting via Transient Storage (EIP-1153)**: Token movements are tracked as net deltas in transient storage throughout an atomic operation. Tokens only physically transfer at settlement, eliminating intermediate ERC-20 transfers on every hop of a multi-leg trade.

Key improvements over Uniswap v3:

| Feature | v3 | v4 |
|---|---|---|
| Pool architecture | One contract per (token0, token1, fee) | Singleton: all pools in `PoolManager` |
| Extensibility | None (immutable logic) | Hooks at every lifecycle event |
| Native ETH | WETH wrapper required | First-class `Currency` type (address(0) = ETH) |
| Fee model | Fixed per pool at creation | Static or hook-controlled dynamic fees |
| Flash accounting | Transfer on each swap step | Net-delta settlement via EIP-1153 transient storage |
| LP positions | ERC-721 NFT (NonfungiblePositionManager) | ERC-721 NFT (PositionManager, periphery) |
| Internal token accounting | None (ERC-20 only) | ERC-6909 claims for gas-optimized internal transfers |
| Position salt | Not supported | `salt` field enables multiple positions per range |
| Donate | Not supported | `donate()` sends tokens directly to in-range LPs |
| Solidity | 0.7.6 | 0.8.24+ (uses `TSTORE`/`TLOAD`) |

The contracts are non-upgradeable. Protocol governance controls the `protocolFeeController` address and can enable fee collection on individual pools.

---

## 2. Architecture & Contract Hierarchy

```
v4-core
├── PoolManager                          [singleton — owns all pool state + tokens]
│   ├── initialize(key, sqrtPriceX96)
│   ├── modifyLiquidity(key, params, hookData) → (BalanceDelta, BalanceDelta)
│   ├── swap(key, params, hookData)      → (BalanceDelta, BalanceDelta)
│   ├── donate(key, amount0, amount1, hookData) → BalanceDelta
│   ├── unlock(data)                     → bytes   [EIP-1153 gating]
│   ├── settle()                         → uint256 [pay ETH/ERC-20 into pool]
│   ├── settleFor(recipient)
│   ├── take(currency, to, amount)       [pull tokens out of pool]
│   ├── mint(to, id, amount)             [ERC-6909 claim mint]
```

```
│   ├── burn(from, id, amount)          [ERC-6909 claim burn]
│   ├── sync(currency)                  [prepare for native settlement]
│   ├── setProtocolFee(key)
│   ├── collectProtocolFees(recipient, currency, amount)
│   └── extsload / exttload             [external storage readers for StateLibrary]
│
├── Pool.sol                            [library — pure pool logic]
│   ├── initialize(...)
│   ├── swap(...)
│   ├── modifyLiquidity(...)
│   └── donate(...)
│
├── Hooks.sol                           [library — hook validation & dispatch]
├── Position.sol                        [library — position accounting]
├── Tick.sol                            [library — tick state management]
├── TickBitmap.sol                      [library — bitmap operations]
├── TickMath.sol                        [library — tick ↔ sqrtPrice]
├── SqrtPriceMath.sol                   [library — price-to-amount math]
├── SwapMath.sol                        [library — single-step swap computation]
├── FullMath.sol                        [library — 512-bit mulDiv]
├── ProtocolFeeLibrary.sol              [library — fee validation]
├── StateLibrary.sol                    [library — read pool state externally]
│
v4-periphery
├── PositionManager                     (ERC-721)
│   ├── modifyLiquidities(unlockData, deadline)
│   ├── modifyLiquiditiesWithoutUnlock(actions, params)
│   ├── Actions: MINT_POSITION, INCREASE_LIQUIDITY, DECREASE_LIQUIDITY,
│   │            BURN_POSITION, COLLECT_FEES, SETTLE, TAKE, SWEEP...
│   └── positions[tokenId] → PositionInfo
│
├── V4Router
│   ├── exactInputSingle(ExactInputSingleParams)
│   ├── exactInput(ExactInputParams)
│   ├── exactOutputSingle(ExactOutputSingleParams)
│   └── exactOutput(ExactOutputParams)
│
└── lens/
    ├── Quoter
    └── StateView
```

---

## 3. Data Structures & Storage Layout

### 3.1 `Slot0` — Packed Pool Header (Value Type)

V4 uses a **value type** (`type Slot0 is bytes32`) rather than a struct, enforcing type safety while retaining bit packing. Access is through a dedicated library:

```
// types/Slot0.sol
type Slot0 is bytes32;

// Bit layout:
// [0..159]   sqrtPriceX96  — uint160, Q64.96 sqrt price
// [160..183] tick          — int24, current tick
// [184..207] protocolFee   — uint24 (high 12 bits = fee1, low 12 bits = fee0)
```

```
// [208..231] lpFee         — uint24 (the static LP fee, or 0x800000 flag for dynamic)
// [232..255] (unused)      — 24 bits padding
```

| Bits | Field | Width | Notes |
|---|---|---|---|
| 0–159 | `sqrtPriceX96` | 160 bits | Q64.96 fixed-point, same as v3 |
| 160–183 | `tick` | 24 bits | Signed, `[MIN_TICK, MAX_TICK]` |
| 184–207 | `protocolFee` | 24 bits | High 12 bits: fee1, low 12 bits: fee0 |
| 208–231 | `lpFee` | 24 bits | Static fee in ppm, or `DYNAMIC_FEE_FLAG` |
| 232–255 | (reserved) | 24 bits | Zeroed |

Getter/setter pattern using assembly:

```
library Slot0Library {
    uint160 internal constant SQRT_PRICE_MASK = type(uint160).max;

    function sqrtPriceX96(Slot0 _packed) internal pure returns (uint160 _sqrtPriceX96) {
        assembly { _sqrtPriceX96 := and(_packed, SQRT_PRICE_MASK) }
    }

    function tick(Slot0 _packed) internal pure returns (int24 _tick) {
        assembly { _tick := sar(160, shl(72, _packed)) }  // sign-extend 24-bit value
    }

    function protocolFee(Slot0 _packed) internal pure returns (uint24 _protocolFee) {
        assembly { _protocolFee := and(shr(184, _packed), 0xFFFFFF) }
    }

    function lpFee(Slot0 _packed) internal pure returns (uint24 _lpFee) {
        assembly { _lpFee := and(shr(208, _packed), 0xFFFFFF) }
    }
}
```

### 3.2 `Pool.State` — Per-Pool Storage

All pool state is held inside `PoolManager` in a mapping keyed by `PoolId`:

```
// Pool.sol
struct State {
    Slot0 slot0;
    uint256 feeGrowthGlobal0X128;   // Q128.128, fee per unit liquidity (token0)
    uint256 feeGrowthGlobal1X128;   // Q128.128, fee per unit liquidity (token1)
    uint128 liquidity;              // active virtual liquidity
    mapping(int24 => TickInfo)      ticks;
    mapping(int16 => uint256)       tickBitmap;
    mapping(bytes32 => Position.State) positions;
}

// In PoolManager:
mapping(PoolId id => Pool.State) internal _pools;
```

The `_pools` mapping is at storage slot 6 in `PoolManager`. The `StateLibrary` derives per-pool storage slots using:

```
// StateLibrary.sol
uint256 public constant POOLS_SLOT = 6;

function _getPoolStateSlot(PoolId poolId) internal pure returns (bytes32) {
    return keccak256(abi.encodePacked(PoolId.unwrap(poolId), POOLS_SLOT));
}
```

### 3.3 `TickInfo` — Per-Tick State

```
// Pool.sol
struct TickInfo {
    uint128 liquidityGross;
    int128  liquidityNet;
    uint256 feeGrowthOutside0X128;
    uint256 feeGrowthOutside1X128;
}
```

Compared to v3, the oracle fields (`tickCumulativeOutside`, `secondsPerLiquidityOutsideX128`, `secondsOutside`) have been removed from tick state. V4 does not include a built-in TWAP oracle — this functionality is delegated to hooks.

### 3.4 `Position.State` — Per-Position State

```
// libraries/Position.sol
struct State {
    uint128 liquidity;
    uint256 feeGrowthInside0LastX128;
    uint256 feeGrowthInside1LastX128;
}
```

Position key derivation now includes a `salt` parameter, allowing a single owner to hold multiple positions with identical tick ranges:

```
function calculatePositionKey(
    address owner,
    int24   tickLower,
    int24   tickUpper,
    bytes32 salt
) internal pure returns (bytes32 positionKey) {
    assembly {
        mstore(0x26, salt)
        mstore(0x06, tickUpper)
        mstore(0x03, tickLower)
        mstore(0,    owner)
        positionKey := keccak256(0, 0x46)
    }
}
```

### 3.5 `BalanceDelta` — Packed Token Amounts

```
// types/BalanceDelta.sol
type BalanceDelta is int256;

// High 128 bits = amount0 (token0 delta)
// Low  128 bits = amount1 (token1 delta)

function amount0(BalanceDelta bd) internal pure returns (int128 _amount0) {
    assembly { _amount0 := sar(128, bd) }
}

function amount1(BalanceDelta bd) internal pure returns (int128 _amount1) {
    assembly { _amount1 := signextend(15, bd) }
}

function toBalanceDelta(int128 a0, int128 a1) pure returns (BalanceDelta bd) {
    assembly {
        bd := or(shl(128, a0), and(a1, 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF))
    }
}
```

This allows a single 256-bit return value to encode both token deltas, saving calldata and stack space.

### 3.6 Transient Storage Variables

V4 introduces transient state (erased after each transaction) for the accounting system:

```
// Stored in transient storage (EIP-1153):
// - Lock state: whether the pool is unlocked
// - Currency deltas: mapping(address locker → mapping(Currency → int256))
// - NonzeroDeltaCount: uint256 count of unsettled currency deltas
// - Currency reserves: used for native ETH sync-settle

// Library: TransientStateLibrary.sol
function currencyDelta(IPoolManager manager, address caller, Currency currency)
    internal view returns (int256)
{
    bytes32 slot = _currencyDeltaSlot(caller, currency);
    return int256(uint256(manager.exttload(slot)));
}
```

---

## 4. Singleton PoolManager & Flash Accounting

### 4.1 The Singleton Pattern

In v2/v3, each pool is a separate contract that holds its own tokens. A multi-hop swap ETH → USDC → DAI requires:

1. Transfer WETH to Pool1
2. Pool1 transfers USDC to Pool2
3. Pool2 transfers DAI to user

In v4, all pools share one contract. The same swap becomes:

1. `PoolManager.unlock()` — set transient lock
2. `swap(ETH→USDC)` — update internal state, record `+ETH delta, -USDC delta`

3. `swap(USDC→DAI)` — update internal state, record `+USDC delta, −DAI delta`
4. Net settle: user transfers ETH in, receives DAI out (USDC deltas cancel out)
5. `NonzeroDeltaCount == 0` → unlock completes

**Gas savings:** Intermediate ERC-20 transfers (steps 2–3 in v3) are eliminated entirely. This saves ~20,000 gas per intermediate hop.

### 4.2 `unlock()` — The Gate Function

```
// PoolManager.sol
function unlock(bytes calldata data) external override returns (bytes memory result) {
    if (Lock.isUnlocked()) UnlockAlreadyUnlocked.selector.revertWith();

    Lock.unlock();

    // Delegate execution to the caller
    result = IUnlockCallback(msg.sender).unlockCallback(data);

    // All currency deltas must be zeroed out before re-locking
    if (NonzeroDeltaCount.read() != 0) CurrencyNotSettled.selector.revertWith();

    Lock.lock();
}
```

`Lock` stores its state in transient storage slot `LOCK_SLOT = keccak256("PoolManager.Lock")`:

```
library Lock {
    bytes32 internal constant IS_UNLOCKED_SLOT =
        0x0aedd6bde10e3aa2adec092b02a3e3e805795516cda41f27aa145b8f300af87a;

    function unlock() internal {
        assembly { tstore(IS_UNLOCKED_SLOT, true) }
    }

    function lock() internal {
        assembly { tstore(IS_UNLOCKED_SLOT, false) }
    }

    function isUnlocked() internal view returns (bool unlocked) {
        assembly { unlocked := tload(IS_UNLOCKED_SLOT) }
    }
}
```

### 4.3 Delta Accounting

Every operation that changes token obligations updates a per-(caller, currency) transient delta:

```
// PoolManager.sol (internal)
function _accountDelta(Currency currency, int128 delta, address target) internal {
    if (delta == 0) return;

    int256 current  = CurrencyDelta.get(target, currency);
    int256 next     = current + delta;
```

```
    unchecked {
        // Track count of non-zero deltas (used for settlement check)
        if (next == 0) {
            NonzeroDeltaCount.decrement();
        } else if (current == 0) {
            NonzeroDeltaCount.increment();
        }
    }

    CurrencyDelta.set(target, currency, next);
}
```

`CurrencyDelta` uses `TSTORE`/`TLOAD` with a slot derived as:

```
function _computeSlot(address target, Currency currency) internal pure returns (bytes32
slot) {
    assembly {
        mstore(0,  target)
        mstore(32, currency)
        slot := keccak256(0, 64)
    }
}
```

### 4.4 Settlement Functions

After operations are complete, the caller must settle all outstanding deltas:

```
// Pay into pool (settles a positive delta for `msg.sender`)
function settle() external payable override onlyWhenUnlocked returns (uint256 paid) {
    return _settle(msg.sender);
}

function settleFor(address recipient) external payable override onlyWhenUnlocked returns
(uint256 paid) {
    return _settle(recipient);
}

// Pull tokens out of pool (settles a negative delta for `to`)
function take(Currency currency, address to, uint256 amount) external override
onlyWhenUnlocked {
    unchecked {
        _accountDelta(currency, amount.toInt128(), msg.sender);  // increment caller's owed
    }
    currency.transfer(to, amount);
}
```

For ERC-20 settlement, the caller transfers tokens to `PoolManager` then calls `settle()`. For native ETH, `sync()` must be called first to snapshot the balance before the ETH transfer:

```
function sync(Currency currency) external {
    // Store current balance in transient storage as "reserved" amount
```

```
    CurrencyReserves.syncCurrencyAndReserves(currency, currency.balanceOfSelf());
}
```

---

## 5. Currency Type & Native ETH Support

### 5.1 `Currency` Value Type

```
// types/Currency.sol
type Currency is address;

library CurrencyLibrary {
    // address(0) represents native ETH
    Currency public constant ADDRESS_ZERO = Currency.wrap(address(0));

    function isAddressZero(Currency currency) internal pure returns (bool) {
        return Currency.unwrap(currency) == address(0);
    }

    function isNative(Currency currency) internal pure returns (bool) {
        return Currency.unwrap(currency) < address(1);
    }

    function transfer(Currency currency, address to, uint256 amount) internal {
        if (isAddressZero(currency)) {
            // Native ETH transfer
            assembly {
                let success := call(gas(), to, amount, 0, 0, 0, 0)
                if iszero(success) {
                    mstore(0, 0x90b8ec18)  // NativeTransferFailed()
                    revert(0x1c, 0x04)
                }
            }
        } else {
            // ERC-20 transfer
            TransferHelper.safeTransfer(Currency.unwrap(currency), to, amount);
        }
    }

    function balanceOfSelf(Currency currency) internal view returns (uint256) {
        if (isAddressZero(currency)) {
            return address(this).balance;
        } else {
            return IERC20Minimal(Currency.unwrap(currency)).balanceOf(address(this));
        }
    }

    // Convert Currency to ERC-6909 claim id
    function toId(Currency currency) internal pure returns (uint256) {
        return uint256(uint160(Currency.unwrap(currency)));
    }
}
```

### 5.2 Token Ordering

As in v3, `currency0 < currency1` by address value. Since `address(0)` is the smallest possible address, **native ETH is always `currency0`** whenever it is part of a pair. This is deterministic and requires no special-casing in path logic.

---

## 6. PoolKey, PoolId & Pool Identification

### 6.1 `PoolKey`

```
// types/PoolKey.sol
struct PoolKey {
    Currency currency0;   // lower address token (address(0) = native ETH)
    Currency currency1;   // higher address token
    uint24   fee;         // LP fee in ppm (0x800000 = dynamic fee managed by hook)
    int24    tickSpacing; // must match feeAmountTickSpacing[fee], or any if hook dynamic
    IHooks   hooks;       // hook contract address (encodes permissions in lower bits)
}
```

The `PoolKey` is the complete pool identifier. Two pools are distinct if **any** field differs — this enables multiple pools with identical token pairs but different fees, tick spacings, or hooks.

### 6.2 `PoolId`

```
// types/PoolId.sol
type PoolId is bytes32;

function toId(PoolKey memory poolKey) pure returns (PoolId poolId) {
    assembly { poolId := keccak256(poolKey, 0xa0) }  // hash all 5 fields (5 × 32 = 160
bytes)
}
```

`PoolId` is used as the key into `PoolManager._pools` and for all hook dispatch events.

### 6.3 Fee Amount to Tick Spacing Registry

```
// PoolManager.sol
mapping(uint24 fee => int24 tickSpacing) public feeAmountTickSpacing;

constructor(address initialOwner) {
    // ... owner setup

    feeAmountTickSpacing[100]   = 1;
    feeAmountTickSpacing[500]   = 10;
    feeAmountTickSpacing[3000]  = 60;
    feeAmountTickSpacing[10000] = 200;
}
```

Hooks using dynamic fees (`fee = 0x800000`) can specify any `tickSpacing` in their `PoolKey`.

---

## 7. Hook System: Architecture & Permissions

### 7.1 Hook Address Encoding

Hook permissions are encoded in the **lowest 14 bits** of the hook contract's address. This means a valid hook contract must be deployed at an address whose lower bits match its intended permissions — achieved via `CREATE2` with a salt search (vanity mining).

```
// libraries/Hooks.sol
uint160 internal constant ALL_HOOK_MASK = uint160((1 << 14) - 1);  // 0x3FFF

// Permission flags (bit position in hook address):
uint160 internal constant BEFORE_INITIALIZE_FLAG                 = 1 << 13;
uint160 internal constant AFTER_INITIALIZE_FLAG                  = 1 << 12;
uint160 internal constant BEFORE_ADD_LIQUIDITY_FLAG              = 1 << 11;
uint160 internal constant AFTER_ADD_LIQUIDITY_FLAG               = 1 << 10;
uint160 internal constant BEFORE_REMOVE_LIQUIDITY_FLAG           = 1 << 9;
uint160 internal constant AFTER_REMOVE_LIQUIDITY_FLAG            = 1 << 8;
uint160 internal constant BEFORE_SWAP_FLAG                       = 1 << 7;
uint160 internal constant AFTER_SWAP_FLAG                        = 1 << 6;
uint160 internal constant BEFORE_DONATE_FLAG                     = 1 << 5;
uint160 internal constant AFTER_DONATE_FLAG                      = 1 << 4;
uint160 internal constant BEFORE_SWAP_RETURNS_DELTA_FLAG         = 1 << 3;
uint160 internal constant AFTER_SWAP_RETURNS_DELTA_FLAG          = 1 << 2;
uint160 internal constant AFTER_ADD_LIQUIDITY_RETURNS_DELTA_FLAG  = 1 << 1;
uint160 internal constant AFTER_REMOVE_LIQUIDITY_RETURNS_DELTA_FLAG = 1 << 0;
```

Permission check at pool initialization:

```
function validateHookPermissions(IHooks self, Permissions memory permissions) internal pure
{
    if (
        permissions.beforeInitialize != self.hasPermission(BEFORE_INITIALIZE_FLAG) ||
        permissions.afterInitialize  != self.hasPermission(AFTER_INITIALIZE_FLAG)  ||
        // ... all 14 flags checked
    ) {
        revert HookAddressNotValid(address(self));
    }
}

function hasPermission(IHooks self, uint160 flag) internal pure returns (bool) {
    return uint160(address(self)) & flag != 0;
}
```

### 7.2 The 14 Hook Points

| Hook | Direction | Can Return Delta? | Trigger |
|---|---|---|---|
| `beforeInitialize` | → pool | No | Pool creation |
| `afterInitialize` | ← pool | No | After first tick set |
| `beforeAddLiquidity` | → pool | No | Before mint |
| `afterAddLiquidity` | ← pool | Yes | After mint |
| `beforeRemoveLiquidity` | → pool | No | Before burn |
| `afterRemoveLiquidity` | ← pool | Yes | After burn |

| Hook | Direction | Can Return Delta? | Trigger |
|---|---|---|---|
| `beforeSwap` | → pool | Yes | Before swap step |
| `afterSwap` | ← pool | Yes | After swap |
| `beforeDonate` | → pool | No | Before donate |
| `afterDonate` | ← pool | No | After donate |

The four "returns delta" hooks (bits 3–0) allow hooks to directly modify the token amounts involved in operations — enabling custom AMM curves, fee redistribution, and synthetic positions.

### 7.3 Hook Dispatch Logic

```
// libraries/Hooks.sol
function beforeSwap(
    IHooks self,
    PoolKey memory key,
    IPoolManager.SwapParams memory params,
    bytes calldata hookData
) internal returns (int256 amountToSwap, BeforeSwapDelta hookReturn, uint24 lpFeeOverride) {
    amountToSwap = params.amountSpecified;

    if (self.hasPermission(BEFORE_SWAP_FLAG)) {
        bytes memory result = callHook(self, abi.encodeCall(
            IHooks.beforeSwap, (msg.sender, key, params, hookData)
        ));
        // Decode returned values
        (bytes4 selector, BeforeSwapDelta returnDelta, uint24 fee) =
            abi.decode(result, (bytes4, BeforeSwapDelta, uint24));

        require(selector == IHooks.beforeSwap.selector, InvalidHookResponse());

        // Apply hook delta override if BEFORE_SWAP_RETURNS_DELTA_FLAG set
        if (self.hasPermission(BEFORE_SWAP_RETURNS_DELTA_FLAG)) {
            int128 hookDeltaSpecified = returnDelta.getSpecifiedDelta();
            if (hookDeltaSpecified != 0) {
                amountToSwap += hookDeltaSpecified;
                self.accountPoolBalanceDelta(key, toBalanceDelta(/*...*/), msg.sender);
            }
        }

        // Dynamic fee override
        if (key.fee.isDynamicFee() && fee != 0) lpFeeOverride = fee;
    }
}
```

The `callHook` function wraps the external call with explicit revert forwarding:

```
function callHook(IHooks self, bytes memory data) internal returns (bytes memory result) {
    bool success;
    assembly {
        success := call(gas(), self, 0, add(data, 0x20), mload(data), 0, 0)
        // copy returndata
        let ptr := mload(0x40)
```

```
        returndatacopy(ptr, 0, returndatasize())
        result := ptr
        mstore(0x40, add(ptr, returndatasize()))
    }
    if (!success) {
        assembly { revert(add(result, 0x20), mload(result)) }
    }
}
```

---

## 8. Hook Interface & Return Deltas

### 8.1 `IHooks` Interface

```
interface IHooks {
    function beforeInitialize(address sender, PoolKey calldata key, uint160 sqrtPriceX96)
        external returns (bytes4);

    function afterInitialize(address sender, PoolKey calldata key, uint160 sqrtPriceX96,
int24 tick)
        external returns (bytes4);

    function beforeAddLiquidity(
        address sender, PoolKey calldata key,
        IPoolManager.ModifyLiquidityParams calldata params, bytes calldata hookData
    ) external returns (bytes4);

    function afterAddLiquidity(
        address sender, PoolKey calldata key,
        IPoolManager.ModifyLiquidityParams calldata params,
        BalanceDelta delta, BalanceDelta feesAccrued, bytes calldata hookData
    ) external returns (bytes4, BalanceDelta);

    function beforeRemoveLiquidity(
        address sender, PoolKey calldata key,
        IPoolManager.ModifyLiquidityParams calldata params, bytes calldata hookData
    ) external returns (bytes4);

    function afterRemoveLiquidity(
        address sender, PoolKey calldata key,
        IPoolManager.ModifyLiquidityParams calldata params,
        BalanceDelta delta, BalanceDelta feesAccrued, bytes calldata hookData
    ) external returns (bytes4, BalanceDelta);

    function beforeSwap(
        address sender, PoolKey calldata key,
        IPoolManager.SwapParams calldata params, bytes calldata hookData
    ) external returns (bytes4, BeforeSwapDelta, uint24);

    function afterSwap(
        address sender, PoolKey calldata key,
        IPoolManager.SwapParams calldata params,
        BalanceDelta delta, bytes calldata hookData
    ) external returns (bytes4, int128);

    function beforeDonate(
        address sender, PoolKey calldata key,
```

```
        uint256 amount0, uint256 amount1, bytes calldata hookData
    ) external returns (bytes4);

    function afterDonate(
        address sender, PoolKey calldata key,
        uint256 amount0, uint256 amount1, bytes calldata hookData
    ) external returns (bytes4);
}
```

All hook functions must return a `bytes4` selector matching the function signature — this is verified by the dispatcher and prevents accidental acceptance of malformed hook responses.

### 8.2 `BeforeSwapDelta`

```
// types/BeforeSwapDelta.sol
type BeforeSwapDelta is int256;

// High 128 bits: delta on the "specified" token (the one the user specified amount for)
// Low  128 bits: delta on the "unspecified" token

function getSpecifiedDelta(BeforeSwapDelta bd) internal pure returns (int128 delta) {
    assembly { delta := sar(128, bd) }
}

function getUnspecifiedDelta(BeforeSwapDelta bd) internal pure returns (int128 delta) {
    assembly { delta := signextend(15, bd) }
}
```

A `beforeSwap` hook returning a non-zero `BeforeSwapDelta` can:

- Partially or fully fill the swap from its own reserves
- Offset the swap amount (e.g., apply a rebate)
- Implement custom AMM curves that run before the CLAMM core

---

## 9. Unlock / Callback Pattern (EIP-1153 Transient Storage)

### 9.1 EIP-1153 Opcodes

EIP-1153, included in the Cancun hard fork (March 2024), introduces two new opcodes:

- `TSTORE(slot, value)` — write to transient storage (cleared at end of transaction)
- `TLOAD(slot)` → value — read from transient storage

Transient storage costs **100 gas per access** (versus 2,100+ for cold persistent SLOAD/SSTORE), making delta tracking economical.

### 9.2 `NonzeroDeltaCount`

```
// libraries/NonzeroDeltaCount.sol
library NonzeroDeltaCount {
    bytes32 internal constant NONZERO_DELTA_COUNT_SLOT =
        0x7d4b3164c6e45b97e7d87b7125a44c5828d005b88297ec8f542c8e2DF2be8b2;
```

```
    function read() internal view returns (uint256 count) {
        assembly { count := tload(NONZERO_DELTA_COUNT_SLOT) }
    }

    function increment() internal {
        assembly {
            let count := tload(NONZERO_DELTA_COUNT_SLOT)
            tstore(NONZERO_DELTA_COUNT_SLOT, add(count, 1))
        }
    }

    function decrement() internal {
        assembly {
            let count := tload(NONZERO_DELTA_COUNT_SLOT)
            tstore(NONZERO_DELTA_COUNT_SLOT, sub(count, 1))
        }
    }
}
```

The `unlock()` function enforces that `NonzeroDeltaCount == 0` after the callback returns, meaning every pending delta was settled.

### 9.3 Typical Unlock Flow

```
User (or Router) → PoolManager.unlock(data)
    └→ IUnlockCallback(msg.sender).unlockCallback(data)
            ├─ PoolManager.swap(key, params, hookData)
            │       → feeGrowth updates, tick crossings, delta accumulation
            ├─ PoolManager.take(currency1, user, amountOut)
            │       → transfers tokens out, increases user's delta for currency1
            └─ PoolManager.settle{value: amountIn}()
                    → ETH received, decreases user's delta for currency0
    └─ NonzeroDeltaCount == 0? → unlock completes ✓
```

For ERC-20:

```
└─ token.transferFrom(user, poolManager, amountIn)  // user approves poolManager
└─ PoolManager.settle()                              // records payment
```

---

## 10. Pool: Swap Mechanics

### 10.1 Swap Function

```
// PoolManager.sol
function swap(
    PoolKey memory key,
    IPoolManager.SwapParams memory params,
    bytes calldata hookData
) external override onlyWhenUnlocked noDelegateCall returns (BalanceDelta swapDelta,
BalanceDelta hookDelta)
{
```

```
    // Validate tick spacing
    if (params.zeroForOne ? params.sqrtPriceLimitX96 >= slot0.sqrtPriceX96
                          : params.sqrtPriceLimitX96 <= slot0.sqrtPriceX96)
    {
        revert InvalidSqrtPrice(params.sqrtPriceLimitX96);
    }

    PoolId id = key.toId();
    Pool.State storage pool = _getPool(id);

    // beforeSwap hook (may modify amountSpecified and/or apply beforeSwapDelta)
    (int256 amountToSwap, BeforeSwapDelta beforeSwapHookDelta, uint24 lpFeeOverride) =
        key.hooks.beforeSwap(key, params, hookData);

    // Execute core swap
    Pool.SwapResult memory result = pool.swap(
        Pool.SwapParams({
            tickSpacing:          key.tickSpacing,
            zeroForOne:           params.zeroForOne,
            amountSpecified:      amountToSwap,
            sqrtPriceLimitX96:    params.sqrtPriceLimitX96,
            lpFeeOverride:        lpFeeOverride
        })
    );

    swapDelta = result.swapDelta;

    // afterSwap hook (may return delta adjustment)
    BalanceDelta afterSwapHookDelta;
    (swapDelta, afterSwapHookDelta) = key.hooks.afterSwap(key, params, swapDelta, hookData);
    hookDelta = afterSwapHookDelta.add(beforeSwapHookDelta.toBalanceDelta());

    // Account deltas to caller
    _accountPoolBalanceDelta(key, swapDelta.add(hookDelta), msg.sender);

    emit Swap(id, msg.sender, result.amount0, result.amount1, result.sqrtPriceX96,
              result.liquidity, result.tick, result.fee);
}
```

### 10.2 `SwapParams`

```
struct SwapParams {
    bool    zeroForOne;           // true = token0→token1 (price decreases)
    int256  amountSpecified;      // >0 = exactInput, <0 = exactOutput
    uint160 sqrtPriceLimitX96;   // price cannot cross this boundary
}
```

### 10.3 Core Swap Loop (`Pool.swap`)

The core swap logic in `Pool.sol` is architecturally identical to v3's swap loop (tick navigation, `SwapMath.computeSwapStep`, tick crossing), with these key differences:

1. **No oracle writes** — tick crossings no longer write oracle observations (no built-in TWAP)
2. **`lpFeeOverride`** — dynamic fee from `beforeSwap` hook replaces static `lpFee`
3. **`TickInfo` lacks oracle fields** — smaller struct, lower tick cross gas

4. **Returns** `SwapResult` — structured return instead of multiple output variables

```
struct SwapResult {
    BalanceDelta swapDelta;   // packed (amount0, amount1)
    uint160 sqrtPriceX96;    // final price
    int24   tick;             // final tick
    uint128 liquidity;        // final active liquidity
    uint24  fee;              // actual fee charged (static or dynamic)
}
```

The tick crossing logic:

```
// On tick cross (price reaches tickNext):
int128 liquidityNet = Pool.crossTick(
    self,
    step.tickNext,
    (zeroForOne ? state.feeGrowthGlobalX128 : feeGrowthGlobal0X128),
    (zeroForOne ? feeGrowthGlobal1X128 : state.feeGrowthGlobalX128)
);
// No oracle observation written here (unlike v3)
if (zeroForOne) liquidityNet = -liquidityNet;
state.liquidity = LiquidityMath.addDelta(state.liquidity, liquidityNet);
```

---

# 11. Pool: Liquidity Management (modifyLiquidity)

## 11.1 Unified Mint/Burn Function

V4 consolidates `mint()` and `burn()` into a single `modifyLiquidity()` call. Positive `liquidityDelta` = add; negative = remove.

```
// PoolManager.sol
function modifyLiquidity(
    PoolKey memory key,
    IPoolManager.ModifyLiquidityParams memory params,
    bytes calldata hookData
) external override onlyWhenUnlocked noDelegateCall
  returns (BalanceDelta callerDelta, BalanceDelta feesAccrued)
{
    PoolId id = key.toId();
    Pool.State storage pool = _getPool(id);

    // Before hook
    key.hooks.beforeModifyLiquidity(key, params, hookData);

    // Execute core liquidity update
    BalanceDelta principalDelta;
    (principalDelta, feesAccrued) = pool.modifyLiquidity(
        Pool.ModifyLiquidityParams({
            owner:        msg.sender,
            tickLower:    params.tickLower,
            tickUpper:    params.tickUpper,
            liquidityDelta: params.liquidityDelta.toInt128(),
            tickSpacing:  key.tickSpacing,
```

```
            salt:         params.salt
        })
    );

    callerDelta = principalDelta + feesAccrued;

    // After hook (may return a delta override)
    BalanceDelta hookDelta;
    (callerDelta, hookDelta) = key.hooks.afterModifyLiquidity(key, params, callerDelta,
feesAccrued, hookData);

    // Account final delta to msg.sender
    if (hookDelta != BalanceDeltaLibrary.ZERO_DELTA) {
        _accountPoolBalanceDelta(key, hookDelta, address(key.hooks));
    }
    _accountPoolBalanceDelta(key, callerDelta, msg.sender);

    emit ModifyLiquidity(id, msg.sender, params.tickLower, params.tickUpper,
                         params.liquidityDelta, params.salt);
}
```

### 11.2 `ModifyLiquidityParams`

```
struct ModifyLiquidityParams {
    int24   tickLower;
    int24   tickUpper;
    int256  liquidityDelta;   // int256 at call site, cast to int128 internally
    bytes32 salt;             // enables multiple positions per (owner, tickLower,
tickUpper)
}
```

The `salt` field is the critical new addition. It allows protocols (e.g., the PositionManager) to hold multiple distinct positions for the same liquidity range on behalf of different users, disambiguated by salt.

### 11.3 Fee Collection vs v3

In v3, fees were collected via `burn(0) + collect()`. In v4:

1. `modifyLiquidity` returns `feesAccrued` as a `BalanceDelta`
2. The caller can choose to reinvest or withdraw fees atomically within the same `unlock()` callback
3. No separate `collect()` function exists at the core level — fee handling is up to the periphery

---

## 12. Pool: Donate

`donate()` is a new primitive allowing anyone to deposit tokens directly into a pool, credited proportionally to all in-range LPs via the fee accumulator — without performing a swap.

```
// PoolManager.sol
function donate(
    PoolKey memory key,
    uint256 amount0,
    uint256 amount1,
    bytes calldata hookData
```

```
) external override onlyWhenUnlocked noDelegateCall returns (BalanceDelta delta)
{
    PoolId id = key.toId();
    Pool.State storage pool = _getPool(id);

    key.hooks.beforeDonate(key, amount0, amount1, hookData);

    delta = pool.donate(Pool.DonateParams({
        tickSpacing: key.tickSpacing,
        amount0:     amount0,
        amount1:     amount1
    }));

    _accountPoolBalanceDelta(key, delta, msg.sender);

    key.hooks.afterDonate(key, amount0, amount1, hookData);
}
```

Inside `Pool.donate`, the donated amounts are distributed by directly incrementing `feeGrowthGlobal`:

```
// Pool.sol
function donate(State storage self, DonateParams memory params)
    internal returns (BalanceDelta delta)
{
    if (self.liquidity == 0) NoLiquidityToReceiveFees.selector.revertWith();

    unchecked {
        if (params.amount0 > 0) {
            self.feeGrowthGlobal0X128 += FullMath.mulDiv(params.amount0, FixedPoint128.Q128,
self.liquidity);
        }
        if (params.amount1 > 0) {
            self.feeGrowthGlobal1X128 += FullMath.mulDiv(params.amount1, FixedPoint128.Q128,
self.liquidity);
        }
    }

    delta = toBalanceDelta(-(params.amount0.toInt128()), -(params.amount1.toInt128()));
}
```

Use cases for `donate()`:

- Protocol rewards for LPs
- TWAMM hooks distributing execution fees
- MEV redistribution mechanisms

---

## 13. Fee Accounting: LP, Protocol & Dynamic Fees

### 13.1 LP Fee

The LP fee is stored in `slot0.lpFee` as a `uint24` in parts per million (same scale as v3). During `initialize()`:

```
function initialize(
    PoolKey memory key,
```

```
    uint160 sqrtPriceX96
) external override noDelegateCall returns (int24 tick)
{
    // Validate fee
    if (key.fee >= ProtocolFeeLibrary.MAX_PROTOCOL_FEE && !key.fee.isDynamicFee())
        revert FeeTooLarge();
    if (key.tickSpacing > MAX_TICK_SPACING) revert TickSpacingTooLarge(key.tickSpacing);
    if (key.tickSpacing < MIN_TICK_SPACING) revert TickSpacingTooSmall(key.tickSpacing);
    if (key.currency0 >= key.currency1) revert CurrenciesOutOfOrderOrEqual(...);
    // ...
}
```

### 13.2 Dynamic Fee Flag

```
// libraries/SwapFeeLibrary.sol
uint24 internal constant DYNAMIC_FEE_FLAG = 0x800000;  // bit 23

function isDynamicFee(uint24 self) internal pure returns (bool) {
    return self == DYNAMIC_FEE_FLAG;
}
```

When a pool has `fee = 0x800000`, the `beforeSwap` hook is responsible for returning the actual fee via the `lpFeeOverride` return value. The protocol validates this:

```
// In Hooks.beforeSwap:
if (key.fee.isDynamicFee() && lpFeeOverride != 0) {
    // lpFeeOverride must not set the dynamic fee flag
    if (lpFeeOverride.isDynamicFee()) InvalidDynamicFee.selector.revertWith();
    pool.setLpFee(lpFeeOverride);
}
```

### 13.3 Protocol Fee

The protocol fee in v4 is encoded as a `uint24` in `slot0`, split into 12 bits for each token:

```
protocolFee = fee1 (high 12 bits) ++ fee0 (low 12 bits)
```

Each 12-bit field encodes a fractional fee as **1/N** of the LP fee:

```
// ProtocolFeeLibrary.sol
uint16 internal constant MAX_PROTOCOL_FEE = 1000;  // = 0.1%, expressed in ppm
uint256 internal constant PIPS_DENOMINATOR = 1_000_000;

function calculateSwapFee(uint16 protocolFee, uint24 lpFee) internal pure returns (uint24
swapFee) {
    unchecked {
        // Protocol fee is a fraction of the LP fee: protocolFee ppm of swapFee
        // total = lpFee + protocolFee portion
        return uint24(protocolFee) + uint24(
            FullMath.mulDivRoundingUp(lpFee, PIPS_DENOMINATOR - protocolFee,
```

```
PIPS_DENOMINATOR)
        );
    }
}
```

Protocol fees are collected via:

```
function collectProtocolFees(
    address recipient,
    Currency currency,
    uint256 amount
) external override returns (uint256 amountCollected) {
    if (msg.sender != protocolFeeController) revert InvalidCaller();
    // ...
}
```

### 13.4 Fee Growth Calculation

The mechanism is identical to v3:

- Global: `feeGrowthGlobal{0,1}X128` in Q128.128
- Inside: derived from `feeGrowthOutside{0,1}X128` at tick boundaries
- Position: `feeGrowthInside{0,1}LastX128` snapshots at last update

The subtraction intentionally overflows in `uint256` arithmetic, correctly handling wrap-around:

```
tokensOwed = uint128(FullMath.mulDiv(
    feeGrowthInsideX128 - position.feeGrowthInsideLastX128,
    position.liquidity,
    FixedPoint128.Q128
));
```

---

## 14. ERC-6909 Claims System

### 14.1 ERC-6909 Multi-Token Standard

`PoolManager` inherits `ERC6909Claims`, implementing a lightweight multi-token standard where each currency maps to a token ID:

```
// token id = uint256(uint160(address(currency)))
function toId(Currency currency) internal pure returns (uint256) {
    return uint256(uint160(Currency.unwrap(currency)));
}
```

ERC-6909 defines:

```
interface IERC6909 {
    function balanceOf(address owner, uint256 id) external view returns (uint256);
    function allowance(address owner, address spender, uint256 id) external view returns
```

```
(uint256);
    function isOperator(address owner, address spender) external view returns (bool);
    function transfer(address to, uint256 id, uint256 amount) external returns (bool);
    function transferFrom(address from, address to, uint256 id, uint256 amount) external
returns (bool);
    function approve(address spender, uint256 id, uint256 amount) external returns (bool);
    function setOperator(address operator, bool approved) external returns (bool);
}
```

### 14.2 Claims Lifecycle

Instead of physically withdrawing ERC-20 tokens from `PoolManager` after every operation, users can **claim** an internal balance:

```
// PoolManager.sol — mint an ERC-6909 claim for currency
function mint(address to, uint256 id, uint256 amount) external override onlyWhenUnlocked {
    unchecked {
        // Deduct from the caller's pending delta
        _accountDelta(CurrencyLibrary.fromId(id), -(amount.toInt128()), msg.sender);
    }
    _mint(to, id, amount);
}

// Burn a claim to settle an outstanding delta
function burn(address from, uint256 id, uint256 amount) external override onlyWhenUnlocked {
    _accountDelta(CurrencyLibrary.fromId(id), amount.toInt128(), msg.sender);
    _burnFrom(from, msg.sender, id, amount);
}
```

Use cases:

- **Gas optimization**: LPs that frequently rebalance avoid repeated ERC-20 withdrawals
- **Composability**: Claims can be transferred between contracts as part of a single transaction
- **Hooks**: Hooks can mint/burn claims for custom accounting logic

### 14.3 Internal vs External Balances

```
User's perspective:
  "real" ERC-20 balance  +  ERC-6909 claim in PoolManager  =  total accessible balance

PoolManager's perspective:
  ERC-20 held by contract  =  sum of all outstanding claims + active LP collateral
```

---

## 15. Periphery: PositionManager & Actions

### 15.1 Architecture

`PositionManager` replaces v3's `NonfungiblePositionManager`. It wraps liquidity positions as ERC-721 NFTs and interacts with `PoolManager` via the unlock/callback pattern using an **actions batch** encoding:

```
// PositionManager.sol
function modifyLiquidities(
    bytes calldata unlockData,   // encoded actions + parameters
    uint256 deadline
) external payable isNotLocked checkDeadline(deadline) {
    _executeActions(unlockData);
}

// Internal: called as IUnlockCallback.unlockCallback
function unlockCallback(bytes calldata data)
    external onlyPoolManager returns (bytes memory)
{
    (bytes calldata actions, bytes[] calldata params) = abi.decode(data, (bytes, bytes[]));
    _dispatch(actions, params);
    return "";
}
```

### 15.2 Action System

Actions are encoded as a `bytes` array where each byte is an opcode:

```
// Actions.sol
uint8 constant INCREASE_LIQUIDITY   = 0x00;
uint8 constant DECREASE_LIQUIDITY   = 0x01;
uint8 constant MINT_POSITION        = 0x02;
uint8 constant BURN_POSITION        = 0x03;

uint8 constant SETTLE               = 0x10;
uint8 constant SETTLE_PAIR          = 0x11;
uint8 constant TAKE                 = 0x12;
uint8 constant TAKE_PAIR            = 0x13;
uint8 constant TAKE_PORTION         = 0x14;
uint8 constant SETTLE_TAKE_PAIR     = 0x15;
uint8 constant CLOSE_CURRENCY       = 0x16;
uint8 constant CLEAR_OR_TAKE        = 0x17;
uint8 constant SWEEP                = 0x18;
```

Dispatching:

```
function _dispatch(bytes calldata actions, bytes[] calldata params) internal {
    uint256 numActions = actions.length;
    for (uint256 actionIndex = 0; actionIndex < numActions; actionIndex++) {
        uint256 action = uint8(actions[actionIndex]);
        if (action == Actions.MINT_POSITION) {
            _mintPosition(params[actionIndex]);
        } else if (action == Actions.INCREASE_LIQUIDITY) {
            _increaseLiquidity(params[actionIndex]);
        } else if (action == Actions.DECREASE_LIQUIDITY) {
            _decreaseLiquidity(params[actionIndex]);
        } else if (action == Actions.BURN_POSITION) {
            _burnPosition(params[actionIndex]);
        } else if (action == Actions.SETTLE) {
            _settle(params[actionIndex]);
        } else if (action == Actions.TAKE) {
```

```
            _take(params[actionIndex]);
        } // ... etc.
    }
}
```

### 15.3 NFT Position Storage

```
// PositionManager.sol
struct PositionInfo {
    // Packed into one slot:
    // bits [0..24]   tickLower    — int24
    // bits [25..48]  tickUpper    — int24
    // bits [49..168] poolId       — truncated to uint120 (lower bits of keccak256)
    // bit  [169]     hasSubscriber — bool (for subscription hooks)
}

mapping(uint256 tokenId => PositionInfo) public positionInfo;
mapping(bytes25 poolId => PoolKey) public poolKeys;  // poolId = lower 25 bytes of PoolId
```

The PositionManager packs tick bounds and a truncated pool ID into a single storage slot — a deliberate gas optimization. Full pool reconstruction uses `poolKeys[bytes25(poolId)]`.

### 15.4 Mint Example

```
// Client-side encoding (simplified):
bytes memory actions = abi.encodePacked(
    uint8(Actions.MINT_POSITION),   // action 1: mint
    uint8(Actions.SETTLE_PAIR)      // action 2: pay tokens
);
bytes[] memory params = new bytes[](2);
params[0] = abi.encode(key, tickLower, tickUpper, liquidity, amount0Max, amount1Max,
recipient, hookData);
params[1] = abi.encode(key.currency0, key.currency1);

positionManager.modifyLiquidities(abi.encode(actions, params), deadline);
```

### 15.5 StateLibrary — External Pool State Reading

Since pool state lives inside `PoolManager`'s private mappings, periphery contracts and off-chain tools use `StateLibrary` (via `extsload`/`exttload`):

```
// StateLibrary.sol
function getSlot0(IPoolManager manager, PoolId poolId)
    internal view returns (uint160 sqrtPriceX96, int24 tick, uint24 protocolFee, uint24
lpFee)
{
    bytes32 stateSlot = _getPoolStateSlot(poolId);  // keccak256(poolId ++ POOLS_SLOT)
    bytes32 data = manager.extsload(stateSlot);     // external storage read

    sqrtPriceX96 = uint160(uint256(data));
    tick         = int24(int256(uint256(data) >> 160));
    protocolFee  = uint24(uint256(data) >> 184);
```

```
    lpFee        = uint24(uint256(data) >> 208);
}

function getLiquidity(IPoolManager manager, PoolId poolId)
    internal view returns (uint128 liquidity)
{
    bytes32 slot = bytes32(uint256(_getPoolStateSlot(poolId)) + LIQUIDITY_OFFSET);
    liquidity = uint128(uint256(manager.extsload(slot)));
}

function getTickInfo(IPoolManager manager, PoolId poolId, int24 tick)
    internal view returns (TickInfo memory info)
{
    bytes32 slot = _getTickInfoSlot(poolId, tick);
    bytes32[] memory data = manager.extsload(slot, 3);  // reads 3 consecutive slots

    info.liquidityGross          = uint128(uint256(data[0]));
    info.liquidityNet            = int128(int256(uint256(data[0]) >> 128));
    info.feeGrowthOutside0X128   = uint256(data[1]);
    info.feeGrowthOutside1X128   = uint256(data[2]);
}
```

`extsload` is exposed by `PoolManager` as a public view function:

```
function extsload(bytes32 slot) external view override returns (bytes32 value) {
    assembly { value := sload(slot) }
}

function extsload(bytes32 startSlot, uint256 nSlots)
    external view override returns (bytes32[] memory values)
{
    values = new bytes32[](nSlots);
    for (uint256 i = 0; i < nSlots; i++) {
        assembly { mstore(add(values, add(0x20, mul(i, 0x20))), sload(add(startSlot, i))) }
    }
}
```

---

# 16. Protocol Constants & Bounds

## 16.1 Core Protocol Constants

| Constant | Value | Description |
|---|---|---|
| `MIN_TICK` | `-887272` | Same as v3 |
| `MAX_TICK` | `887272` | Same as v3 |
| `MIN_SQRT_RATIO` | `4295128739` | `getSqrtRatioAtTick(MIN_TICK)` |
| `MAX_SQRT_RATIO` | `1461446703485210103287273052203988822378723970342` | `getSqrtRatioAtTick(MAX_TICK)` |
| `Q96` | `2^96` | Q64.96 scaling factor |
| `Q128` | `2^128` | Q128.128 fee accumulator scaling |

| Constant | Value | Description |
|---|---|---|
| `DYNAMIC_FEE_FLAG` | `0x800000` | Marks a pool as using hook-managed dynamic fees |
| `MAX_PROTOCOL_FEE` | `1000` (0.1%) | Maximum protocol fee per token |
| `POOLS_SLOT` | `6` | Storage slot of `_pools` mapping in PoolManager |

### 16.2 Hook Permission Flags (Bit Positions in Hook Address)

| Bit | Flag | Constant Value |
|---|---|---|
| 13 | `BEFORE_INITIALIZE` | `0x2000` |
| 12 | `AFTER_INITIALIZE` | `0x1000` |
| 11 | `BEFORE_ADD_LIQUIDITY` | `0x0800` |
| 10 | `AFTER_ADD_LIQUIDITY` | `0x0400` |
| 9 | `BEFORE_REMOVE_LIQUIDITY` | `0x0200` |
| 8 | `AFTER_REMOVE_LIQUIDITY` | `0x0100` |
| 7 | `BEFORE_SWAP` | `0x0080` |
| 6 | `AFTER_SWAP` | `0x0040` |
| 5 | `BEFORE_DONATE` | `0x0020` |
| 4 | `AFTER_DONATE` | `0x0010` |
| 3 | `BEFORE_SWAP_RETURNS_DELTA` | `0x0008` |
| 2 | `AFTER_SWAP_RETURNS_DELTA` | `0x0004` |
| 1 | `AFTER_ADD_LIQUIDITY_RETURNS_DELTA` | `0x0002` |
| 0 | `AFTER_REMOVE_LIQUIDITY_RETURNS_DELTA` | `0x0001` |

A hook contract controlling all 14 flags must be deployed at an address whose lower 14 bits are all 1: `addr & 0x3FFF == 0x3FFF`.

### 16.3 Tick Spacing Bounds

```
int24 internal constant MAX_TICK_SPACING = type(int16).max;  // = 32767
int24 internal constant MIN_TICK_SPACING = 1;
```

### 16.4 Fee Bounds

| Parameter | Bound | Notes |
|---|---|---|
| Max static LP fee | `999,999` ppm (< 100%) | Enforced at initialization |
| Dynamic fee flag | `0x800000` | Exactly equal, not a range |
| Max protocol fee | `1,000` ppm (0.1%) | Each side (fee0, fee1) independently |

| Parameter | Bound | Notes |
|---|---|---|
| Protocol fee bits | 12 bits per token | Range `[0, 4095]` but max = 1000 |

### 16.5 `BalanceDelta` and Liquidity Bounds

| Parameter | Type | Max Value |
|---|---|---|
| `liquidity` | `uint128` | `2^128 - 1 ≈ 3.4 × 10^38` |
| `liquidityDelta` | `int256` → cast `int128` | Must fit in `int128` |
| `BalanceDelta` field | `int128` per token | `2^127 - 1 ≈ 1.7 × 10^38` |
| `sqrtPriceX96` | `uint160` | Bounded by MIN/MAX_SQRT_RATIO |

---

## 17. Notable Design Decisions

### 17.1 Singleton vs Per-Pool Contracts

The singleton `PoolManager` achieves several engineering benefits:

- **Multi-hop efficiency**: ETH/USDC/DAI swap in one unlock requires only 1 ETH transfer and 1 DAI transfer — no intermediate USDC movement
- **Reduced deployment cost**: No factory-per-pair contract deployment, only `initialize()` needed
- **Shared liquidity accounting**: Hooks can theoretically share internal liquidity across pools within one callback
- **Reduced approve fragmentation**: Approving `PoolManager` once covers all pools

The tradeoff is **blast radius**: a critical bug in `PoolManager` would affect all pools simultaneously.

### 17.2 Hook Address Encoding (Vanity Mining)

Embedding permissions in the hook address is an elegant solution to the permission challenge:

- No separate registry or onchain permission check
- Immutable after deployment (can't add permissions post-deploy without redeployment)
- Requires `CREATE2` vanity mining — the hook contract must be deployed at an address with specific lower bits set

```
// Example: a hook needing BEFORE_SWAP + AFTER_SWAP must be at an address where:
// addr & (BEFORE_SWAP_FLAG | AFTER_SWAP_FLAG) == (BEFORE_SWAP_FLAG | AFTER_SWAP_FLAG)
// addr & 0x00C0 == 0x00C0
```

This requires finding a `CREATE2` salt such that:

```
keccak256(0xff ++ deployer ++ salt ++ initCodeHash)[12:] & 0x3FFF == requiredFlags
```

Expected salt search iterations: `2^(popcount(flags))` — exponential in the number of enabled flags.

### 17.3 Removal of the Built-in TWAP Oracle

V4 deliberately removes the on-chain TWAP oracle that existed in v3. Rationale:

- The v3 oracle had high maintenance burden (cardinality expansion, gas for writes)
- Hook-based oracles are more flexible (arbitrary data can be tracked)

- `GeomeanOracle` hook is available as a reference implementation
- TWAP can be accumulated at any granularity or using any data source

This is a notable breaking change for protocols that rely on `IUniswapV3Pool.observe()`.

### 17.4 Transient Storage vs Reentrancy Lock

V3 used `slot0.unlocked` (persistent storage) as a reentrancy guard — an SSTORE/SLOAD at 5,000–20,000 gas. V4 replaces this with `TSTORE`/`TLOAD` at 100 gas each, reducing reentrancy guard overhead by 50–200×.

The semantics are identical but stronger: transient storage is automatically cleared at the end of the transaction, so there is no risk of an aborted transaction leaving the lock permanently set.

### 17.5 `noDelegateCall` on Singleton

As in v3, the `noDelegateCall` modifier protects against malicious delegation of pool operations:

```
// NoDelegateCall.sol
abstract contract NoDelegateCall {
    address private immutable _original;
    constructor() { _original = address(this); }
    modifier noDelegateCall() {
        if (address(this) != _original) revert DelegateCallNotAllowed();
        _;
    }
}
```

This is especially critical in the singleton: a `delegatecall` into `PoolManager` could let an attacker manipulate the balances of all pools simultaneously.

### 17.6 Flash Accounting as a General Execution Model

The `unlock()` model is more general than a simple reentrancy gate — it is effectively a **deferred settlement system**. Any sequence of operations that is net-neutral in token terms can execute atomically without holding token balances at intermediate steps. This enables:

- Flash loans without a dedicated `flash()` function (take tokens via `take()`, return them via `settle()`)
- Atomic multi-pool arbitrage
- LP rebalancing without liquidating positions
- Protocol-level composability (a hook calling back into `PoolManager` within the same unlock)

### 17.7 ERC-6909 Claims vs ERC-20 Withdrawals

The choice to implement ERC-6909 (rather than an ad-hoc internal balance) provides standardized token interfaces for internal claims, enabling:

- Wallets and explorers to display claim balances natively
- Protocols to accept claims as collateral without custom integration
- Seamless interoperability between hooks that produce claims and users that consume them

### 17.8 Backwards Compatibility: v3 Tick Math Preserved

V4 preserves the exact same `TickMath` library from v3, including the 19 magic constant multiplications in `getSqrtRatioAtTick` and the assembly-based log2 in `getTickAtSqrtRatio`. This ensures bit-exact compatibility with all off-chain tooling built for v3 tick arithmetic.