

# Uniswap v2: Technical Report

---

**Protocol:** Uniswap v2

**Solidity Version (Core):** =0.5.16

**Solidity Version (Periphery):** =0.6.6

**Scope:** v2-core ([UniswapV2Factory](#), [UniswapV2Pair](#), [UniswapV2ERC20](#)) + v2-periphery ([UniswapV2Router02](#), [UniswapV2Library](#))



**Analyzed by:** rubencrxz

---

## Table of Contents

1. [Protocol Overview](#)
  2. [Architecture & Contract Hierarchy](#)
  3. [Data Structures & Storage Layout](#)
  4. [Constant Product Invariant & Math](#)
  5. [Factory: Pool Deployment & Discovery](#)
  6. [Pair: Swap Mechanics](#)
  7. [Pair: Liquidity Provisioning](#)
  8. [Fee Accounting System](#)
  9. [Price Oracle \(TWAP\)](#)
  10. [Flash Swaps](#)
  11. [UniswapV2ERC20 & Permit](#)
  12. [Periphery: Library & Router](#)
  13. [Non-standard Token Handling](#)
  14. [Protocol Constants & Bounds](#)
  15. [Notable Design Decisions](#)
- 

## 1. Protocol Overview

Uniswap v2 is a constant-product automated market maker (CPAMM) deployed as a system of independent pair contracts, each holding reserves of exactly two ERC-20 tokens and enforcing the invariant  $x \cdot y = k$ . It is the canonical reference implementation of the CPAMM model and the foundation upon which most subsequent AMM designs were built.

The protocol is divided into two layers:

- **Core (v2-core):** A minimal, fund-holding primitive. Stores reserves, enforces the pricing invariant, mints/burns LP tokens, and accumulates the TWAP oracle. Deliberately exposes only low-level functions — `swap()`, `mint()`, `burn()`, `sync()`, `skim()`. Contains no user-intent logic.
- **Periphery (v2-periphery):** User-facing routing layer. Computes trade amounts, enforces slippage bounds, handles ETH/WETH wrapping, chains multi-hop paths, and supports fee-on-transfer tokens. Upgradeable independently of core.

Key improvements over Uniswap v1:

Feature	v1	v2
Token pairs	ERC-20 / ETH only	Any ERC-20 / ERC-20
Price oracle	None (manipulable spot)	Cumulative TWAP accumulators
Flash borrowing	None	Flash swaps (optimistic transfer + callback)
Protocol fee	None	Optional 0.05% (1/6 of LP fee), toggled by <code>feeToSetter</code>
ETH handling	Native	Via WETH (periphery)
Pool addresses	CREATE (non-deterministic)	CREATE2 (deterministic)

Feature	v1	v2
Solidity version	Vyper	Solidity

The contracts are non-upgradeable. The only privileged operation is `setFeeTo` / `setFeeToSetter` on the Factory, which controls the protocol fee recipient but does not affect core invariants.

## 2. Architecture & Contract Hierarchy



The dependency graph is strictly one-directional. Core has no knowledge of Router or Library. Router imports Library for pure computations and calls `Pair.swap()` / `Pair.mint()` / `Pair.burn()` directly.

## 3. Data Structures & Storage Layout

### 3.1 UniswapV2ERC20 (Inherited by Pair)

```

Slot 0: totalSupply          (uint256)
Slot 1: balanceOf           (mapping(address => uint256))
Slot 2: allowance           (mapping(address => mapping(address => uint256)))
Slot 3: DOMAIN_SEPARATOR   (bytes32) – set in constructor, chain-specific
Slot 4: nonces               (mapping(address => uint256))

```

Constants (not in storage):

```

string public constant name    = 'Uniswap V2';
string public constant symbol  = 'UNI-V2';
uint8  public constant decimals = 18;
bytes32 public constant PERMIT_TYPEHASH =
keccak256("Permit(address owner,address spender,uint256 value,uint256 nonce,uint256 deadline)");

```

### 3.2 UniswapV2Pair Storage

```

Slot 5: factory              (address, 20 bytes)
Slot 6: token0                (address, 20 bytes)
Slot 7: token1                (address, 20 bytes)
Slot 8: [packed]
    reserve0            (uint112, bits 0-111)
    reserve1            (uint112, bits 112-223)
    blockTimestampLast (uint32, bits 224-255)
Slot 9: price0CumulativeLast (uint256)
Slot 10: price1CumulativeLast (uint256)
Slot 11: kLast                 (uint256) – reserve0 × reserve1 at last mint/burn
Slot 12: unlocked              (uint256) – reentrancy guard, initialized to 1

```

The critical packing in slot 8 fits exactly one EVM word (256 bits):

- $112 + 112 + 32 = 256$  bits — zero padding, no wasted space.

This design means a single `SLOAD` fetches both reserves and the timestamp in one opcode.

### 3.3 getReserves() — Packed Read

```

function getReserves() public view returns (
    uint112 _reserve0,
    uint112 _reserve1,
    uint32 _blockTimestampLast
) {
    _reserve0      = reserve0;
    _reserve1      = reserve1;
    _blockTimestampLast = blockTimestampLast;
}

```

Because `reserve0`, `reserve1`, and `blockTimestampLast` are packed into the same storage slot, the compiler emits a single `SLOAD` followed by bit-masking to extract each field. This is a deliberate gas optimization — the hot path (every swap) reads reserves exactly once.

### 3.4 UQ112x112 — Fixed-Point Price Encoding

The oracle uses a custom Q112.112 fixed-point library. A `uint224` stores a rational number  $p$  as the integer  $p \times 2^{112}$ , giving 112 bits of integer part and 112 bits of fractional part.

```

library UQ112x112 {
    uint224 constant Q112 = 2**112;

    // Encode uint112 y as UQ112x112: y * 2^112
}

```

```

function encode(uint112 y) internal pure returns (uint224 z) {
    z = uint224(y) * Q112; // never overflows: max is (2^112 - 1) * 2^112 < 2^224
}

// Divide UQ112x112 x by uint112 y, returning UQ112x112
function uqdiv(uint224 x, uint112 y) internal pure returns (uint224 z) {
    z = x / uint224(y);
}
}

```

Price of token0 in token1: `UQ112x112.encode(reserve1).uqdiv(reserve0)` → a `uint224` representing `reserve1 / reserve0` with 112 bits of fractional precision.

The result is then multiplied by `timeElapsed` (a `uint32`) and accumulated into a `uint256`, which can hold up to  $\sim 2^{256}/2^{112} = 2^{144}$  seconds of accumulated price — effectively infinite for practical purposes.

## 4. Constant Product Invariant & Math

### 4.1 Core Invariant

Every Uniswap v2 pool enforces the constant-product invariant:

$$x \cdot y = k$$

where `x = reserve0`, `y = reserve1`, and `k` is a constant that can only increase (via fee accumulation). Trades move the pool along the hyperbola `x · y = k`; liquidity additions/removals shift the hyperbola itself.

### 4.2 Spot Price

The instantaneous marginal price of `token0` in terms of `token1` (ignoring fees) is:

$$\begin{aligned} p_0 &= \text{reserve1} / \text{reserve0} \\ p_1 &= \text{reserve0} / \text{reserve1} \end{aligned}$$

Uniswap v2 tracks both `price0` and `price1` independently in the oracle because arithmetic mean prices are not reciprocals of each other (unlike geometric means).

### 4.3 Fee-Adjusted Swap Invariant

The 0.30% LP fee is enforced by the invariant check at the end of every swap. Rather than computing fee amounts directly, the contract multiplies balances and inputs by 1000 and inputs by 3 (i.e., 0.3% = 3/1000) and checks:

$$(1000 \cdot \text{balance0} - 3 \cdot \text{amount0In}) \cdot (1000 \cdot \text{balance1} - 3 \cdot \text{amount1In}) \geq \text{reserve0} \cdot \text{reserve1} \cdot 1000^2$$

This formulation generalizes correctly to flash swaps where both `amount0In` and `amount1In` may be non-zero (repayment in same token), and is verified in the actual contract as:

```

uint balance0Adjusted = balance0.mul(1000).sub(amount0In.mul(3));
uint balance1Adjusted = balance1.mul(1000).sub(amount1In.mul(3));
require(
    balance0Adjusted.mul(balance1Adjusted) >= uint(_reserve0).mul(_reserve1).mul(1000**2),
    'UniswapV2: K'
);

```

### 4.4 Slippage and Price Impact

For a trade of `amountIn` of token0 receiving `amountOut` of token1 (exact-in, fee-adjusted):

```
amountOut = (amountIn * 997 * reserve1) / (reserve0 * 1000 + amountIn * 997)
```

The implied price impact for a given trade size can be derived by comparing the spot price `reserve1 / reserve0` with the effective execution price `amountOut / amountIn`.

For exact-out (user specifies desired `amountOut`, solves for required `amountIn`):

```
amountIn = ⌈(reserve0 * amountOut * 1000) / ((reserve1 - amountOut) * 997)⌉
```

The ceiling (`+1` after integer division) ensures the invariant is never undershot.

#### 4.5 Babylonian Square Root (Math.sol)

Used in `mint()` for initial LP calculation and `_mintFee()` for protocol fee computation:

```
function sqrt(uint y) internal pure returns (uint z) {
    if (y > 3) {
        z = y;
        uint x = y / 2 + 1;
        while (x < z) {
            z = x;
            x = (y / x + x) / 2;
        }
    } else if (y != 0) {
        z = 1;
    }
}
```

This is an iterative Newton-Raphson (Babylonian) method that converges in  $O(\log \log y)$  iterations. For typical reserve magnitudes it requires ~6–10 iterations.

### 5. Factory: Pool Deployment & Discovery

#### 5.1 State

```
contract UniswapV2Factory {
    address public feeTo;           // protocol fee recipient (address(0) = fee off)
    address public feeToSetter;     // address allowed to change feeTo

    mapping(address => mapping(address => address)) public getPair;
    address[] public allPairs;
}
```

#### 5.2 createPair — CREATE2 Deployment

```
function createPair(address tokenA, address tokenB) external returns (address pair) {
    require(tokenA != tokenB, 'UniswapV2: IDENTICAL_ADDRESSES');
    (address token0, address token1) = tokenA < tokenB
        ? (tokenA, tokenB)
        : (tokenB, tokenA);           // canonical ordering by address value
    require(token0 != address(0), 'UniswapV2: ZERO_ADDRESS');
    require(getPair[token0][token1] == address(0), 'UniswapV2: PAIR_EXISTS');

    bytes memory bytecode = type(UniswapV2Pair).creationCode;
```

```

bytes32 salt = keccak256(abi.encodePacked(token0, token1));
assembly {
    pair := create2(0, add(bytecode, 32), mload(bytecode), salt)
}
IUniswapV2Pair(pair).initialize(token0, token1); // sets token0/token1 post-deploy
getPair[token0][token1] = pair;
getPair[token1][token0] = pair; // bidirectional lookup
allPairs.push(pair);
emit PairCreated(token0, token1, pair, allPairs.length);
}

```

Key design points:

- **Token ordering:** tokens are sorted by address value (`token0 < token1`). This ensures each pair has exactly one canonical address regardless of call order.
- **CREATE2 salt:** `keccak256(token0 ++ token1)` — deterministic given the token pair.
- **initialize():** Because the Pair constructor cannot receive arguments via CREATE2 in Solidity 0.5.x, `token0` and `token1` are set via a post-deploy `initialize()` call restricted to `factory`.

### 5.3 Deterministic Address Derivation

Any contract or off-chain actor can compute a Pair's address without querying the chain:

```

pair_address = keccak256(
    0xff
    ++ factory_address
    ++ keccak256(token0 ++ token1) // salt
    ++ 0x96e8ac4277198ff8b6f785478aa9a39f // init code hash (keccak256 of Pair bytecode)
    403cb768dd02cbee326c3e7da348845f
)[12:]

```

The init code hash `0x96e8ac4...` is a fixed constant derived from the compiled Pair bytecode. This is hardcoded in `UniswapV2Library.pairFor()`. Forks that modify the Pair contract must recompute and update this hash.

### 5.4 Protocol Fee Toggle

```

function setFeeTo(address _feeTo) external {
    require(msg.sender == feeToSetter, 'UniswapV2: FORBIDDEN');
    feeTo = _feeTo;
}

function setFeeToSetter(address _feeToSetter) external {
    require(msg.sender == feeToSetter, 'UniswapV2: FORBIDDEN');
    feeToSetter = _feeToSetter;
}

```

`feeToSetter` is a one-of-a-kind governance key. Setting `feeTo != address(0)` activates the 0.05% protocol fee on all pools. There is no per-pool fee configuration — it is global.

---

## 6. Pair: Swap Mechanics

### 6.1 Reentrancy Guard

All state-changing external functions in `UniswapV2Pair` are protected by a simple mutex:

```

uint private unlocked = 1;

modifier lock() {
    require(unlocked == 1, 'UniswapV2: LOCKED');
}

```

```

    unlocked = 0;
    _;
    unlocked = 1;
}

```

This single `uint` flag costs one `SSTORE` on entry and one on exit (warm slots after EIP-2929: 100 gas each). It prevents reentrancy from ERC-777 hooks, flash swap callbacks, and any other external code paths that might call back into the Pair during execution. All of `swap()`, `mint()`, and `burn()` carry `lock`.

## 6.2 Safe Transfer Wrapper

The Pair never uses `ERC20.transferFrom()`. It only calls `transfer()` directly on token contracts using a hand-rolled safe-transfer wrapper that handles tokens that do not return a boolean:

```

bytes4 private constant SELECTOR = bytes4(keccak256(bytes('transfer(address,uint256)')));

function _safeTransfer(address token, address to, uint value) private {
    (bool success, bytes memory data) = token.call(
        abi.encodeWithSelector(SELECTOR, to, value)
    );
    require(
        success && (data.length == 0 || abi.decode(data, (bool))),
        'UniswapV2: TRANSFER_FAILED'
    );
}

```

The check `data.length == 0 || abi.decode(data, (bool))` handles:

- Compliant ERC-20 tokens (return `true`)
- Non-returning tokens like USDT/BNB (`data.length == 0`)

## 6.3 swap() — Complete Execution Flow

```

function swap(
    uint amount0Out,
    uint amount1Out,
    address to,
    bytes calldata data
) external lock {
    require(amount0Out > 0 || amount1Out > 0, 'UniswapV2: INSUFFICIENT_OUTPUT_AMOUNT');
    (uint128 _reserve0, uint128 _reserve1,) = getReserves();
    require(amount0Out < _reserve0 && amount1Out < _reserve1, 'UniswapV2: INSUFFICIENT_LIQUIDITY');

    uint balance0;
    uint balance1;
    {
        address _token0 = token0;
        address _token1 = token1;
        require(to != _token0 && to != _token1, 'UniswapV2: INVALID_TO');

        // 1. Optimistic output transfer
        if (amount0Out > 0) _safeTransfer(_token0, to, amount0Out);
        if (amount1Out > 0) _safeTransfer(_token1, to, amount1Out);

        // 2. Optional callback (flash swap)
        if (data.length > 0)
            IUniswapV2Callee(to).uniswapV2Call(msg.sender, amount0Out, amount1Out, data);

        // 3. Read post-callback balances (ground truth)
        balance0 = IERC20(_token0).balanceOf(address(this));
        balance1 = IERC20(_token1).balanceOf(address(this));
    }
}

```

```

// 4. Infer inputs from balance deltas
uint amount0In = balance0 > _reserve0 - amount0Out
    ? balance0 - (_reserve0 - amount0Out) : 0;
uint amount1In = balance1 > _reserve1 - amount1Out
    ? balance1 - (_reserve1 - amount1Out) : 0;
require(amount0In > 0 || amount1In > 0, 'UniswapV2: INSUFFICIENT_INPUT_AMOUNT');

// 5. Fee-adjusted invariant check
{
    uint balance0Adjusted = balance0.mul(1000).sub(amount0In.mul(3));
    uint balance1Adjusted = balance1.mul(1000).sub(amount1In.mul(3));
    require(
        balance0Adjusted.mul(balance1Adjusted) >=
        uint(_reserve0).mul(_reserve1).mul(1000**2),
        'UniswapV2: K'
    );
}

// 6. Update reserves and oracle accumulators
_update(balance0, balance1, _reserve0, _reserve1);
emit Swap(msg.sender, amount0In, amount1In, amount0Out, amount1Out, to);
}

```

### Step-by-step:

1. **Optimistic transfer:** Output tokens are sent to `to` before validating payment. This is what enables flash swaps.
2. **Callback:** If `data` is non-empty, `IUniswapV2Callee(to).uniswapV2Call(...)` is invoked. The callee can use the tokens and arrange repayment within this call.
3. **Balance read:** After the callback returns, actual ERC-20 balances are read. This is the "ground truth" — the Pair does not trust any user-provided amount.
4. **Input inference:** `amountIn = max(balance - (reserve - amountOut), 0)`. This computes how much was deposited by comparing current balance against expected balance if no input arrived.
5. **K-check:** The fee-adjusted invariant must hold. If underpaid (or if the callback drained the pool), the transaction reverts.
6. **\_update():** Reserves are updated and oracle accumulators are incremented.

### 6.4 \_update() — Reserve Update & Oracle Accumulation

```

function _update(
    uint balance0,
    uint balance1,
    uint128 _reserve0,
    uint128 _reserve1
) private {
    require(
        balance0 <= uint128(-1) && balance1 <= uint128(-1),
        'UniswapV2: OVERFLOW'
    );
    uint32 blockTimestamp = uint32(block.timestamp % 2**32);
    uint32 timeElapsed = blockTimestamp - blockTimestampLast; // intended overflow

    if (timeElapsed > 0 && _reserve0 != 0 && _reserve1 != 0) {
        // Overflow in the accumulator is intentional and expected
        price0CumulativeLast +=
            uint(UQ112x112.encode(_reserve1).uqdiv(_reserve0)) * timeElapsed;
        price1CumulativeLast +=
            uint(UQ112x112.encode(_reserve0).uqdiv(_reserve1)) * timeElapsed;
    }
    reserve0 = uint128(balance0);
    reserve1 = uint128(balance1);
    blockTimestampLast = blockTimestamp;
    emit Sync(reserve0, reserve1);
}

```

Critical properties:

- **Once-per-block oracle:** `timeElapsed > 0` ensures accumulators are only updated on the *first* interaction per block. All subsequent interactions in the same block see `timeElapsed == 0` and skip the accumulation step. This makes the oracle reflect end-of-previous-block prices, not intra-block manipulated prices.
- **Overflow-safe timestamp:** `blockTimestamp - blockTimestampLast` overflows correctly in `uint32` arithmetic, handling the year 2106 rollover.
- **Price uses cached reserves:** The price added is `_reserve1/_reserve0` (the cached state from before this interaction), not the new post-trade balances. This is the oracle manipulation protection: an attacker who trades to shift reserves cannot affect the oracle for the current block.
- **Overflow intentional in accumulators:** `price0CumulativeLast` and `price1CumulativeLast` are `uint256`. TWAP consumers compute differences, so wrapping overflow is harmless as long as observations are taken within ~136 years.

## 6.5 sync() and skim()

```
// Force reserves to match current balances (recovery for deflating tokens)
function sync() external lock {
    _update(
        IERC20(token0).balanceOf(address(this)),
        IERC20(token1).balanceOf(address(this)),
        reserve0,
        reserve1
    );
}

// Remove surplus balance above reserve (recovery for overflow)
function skim(address to) external lock {
    address _token0 = token0;
    address _token1 = token1;
    _safeTransfer(_token0, to, IERC20(_token0).balanceOf(address(this)).sub(reserve0));
    _safeTransfer(_token1, to, IERC20(_token1).balanceOf(address(this)).sub(reserve1));
}
```

- `sync()` is a recovery valve for tokens that can deflate balances asynchronously (e.g., rebasing tokens). Without it, the reserves would overstate actual holdings permanently.
- `skim()` recovers tokens donated directly to the Pair or overflows above  $2^{112} - 1$ . Without it, swaps would revert on the `OVERFLOW` check in `_update()`.

## 7. Pair: Liquidity Provisioning

### 7.1 mint() — Adding Liquidity

```
function mint(address to) external lock returns (uint liquidity) {
    (uint112 _reserve0, uint112 _reserve1,) = getReserves();
    uint balance0 = IERC20(token0).balanceOf(address(this));
    uint balance1 = IERC20(token1).balanceOf(address(this));
    uint amount0 = balance0.sub(_reserve0); // tokens deposited since last update
    uint amount1 = balance1.sub(_reserve1);

    bool feeOn = _mintFee(_reserve0, _reserve1);
    uint _totalSupply = totalSupply;

    if (_totalSupply == 0) {
        // Initial mint: geometric mean, minus MINIMUM_LIQUIDITY
        liquidity = Math.sqrt(amount0.mul(amount1)).sub(MINIMUM_LIQUIDITY);
        _mint(address(0), MINIMUM_LIQUIDITY); // permanently burned
    } else {
        // Subsequent mints: proportional, take the minimum (limiting side)
        liquidity = Math.min(
            amount0.mul(_totalSupply) / _reserve0,
            amount1.mul(_totalSupply) / _reserve1
        );
    }
}
```

```

    );
}

require(liquidity > 0, 'UniswapV2: INSUFFICIENT_LIQUIDITY_MINTED');
_mint(to, liquidity);

_update(balance0, balance1, _reserve0, _reserve1);
if (feeOn) kLast = uint(reserve0).mul(reserve1);
emit Mint(msg.sender, amount0, amount1);
}

```

## 7.2 Initial LP and MINIMUM\_LIQUIDITY

The first depositor receives  $\sqrt{amount0 * amount1} - 1000$  LP tokens. 1000 units (= `MINIMUM_LIQUIDITY = 10^3`) are permanently minted to `address(0)` and can never be burned.

This solves two problems:

1. **Share price inflation attack:** Without permanently locked liquidity, an attacker could be the first depositor and then donate directly to the Pair to inflate the price per LP share to a level that rounds out small depositors. With 1000 shares locked, raising the value of one share to \$1 would require \$1000 permanently locked — raising it to \$1M would require \$10^9.
2. **Division-by-zero:** Ensures `totalSupply` can never return to zero once a pool has been initialized, preventing arithmetic failures in proportional calculations.

## 7.3 LP Share Proportionality

For subsequent deposits:

```

liquidity = min(
    amount0 * totalSupply / reserve0,
    amount1 * totalSupply / reserve1
)

```

The `min()` ensures that the liquidity minted reflects the limiting token. If a depositor provides an unbalanced ratio (more `token0` than the pool ratio dictates), only the proportionally correct amount is credited as liquidity — the excess `token0` becomes a donation to existing LPs.

The Router avoids this by computing the optimal deposit amounts before calling `mint()`, ensuring the ratio matches reserves.

## 7.4 burn() — Removing Liquidity

```

function burn(address to) external lock returns (uint amount0, uint amount1) {
    (uint112 _reserve0, uint112 _reserve1,) = getReserves();
    address _token0 = token0;
    address _token1 = token1;
    uint balance0 = IERC20(_token0).balanceOf(address(this));
    uint balance1 = IERC20(_token1).balanceOf(address(this));
    uint liquidity = balanceOf[address(this)]; // LP tokens sent to Pair before calling burn()

    bool feeOn = _mintFee(_reserve0, _reserve1);
    uint _totalSupply = totalSupply;

    // Pro-rata redemption using live balances (not cached reserves)
    amount0 = liquidity.mul(balance0) / _totalSupply;
    amount1 = liquidity.mul(balance1) / _totalSupply;
    require(amount0 > 0 && amount1 > 0, 'UniswapV2: INSUFFICIENT_LIQUIDITY_BURNED');

    _burn(address(this), liquidity);
    _safeTransfer(_token0, to, amount0);
    _safeTransfer(_token1, to, amount1);

    balance0 = IERC20(_token0).balanceOf(address(this));
    balance1 = IERC20(_token1).balanceOf(address(this));

    _update(balance0, balance1, _reserve0, _reserve1);
}

```

```

    if (feeOn) kLast = uint(reserve0).mul(reserve1);
    emit Burn(msg.sender, amount0, amount1, to);
}

```

The Router transfers LP tokens to the Pair before calling `burn()`. The Pair reads `balanceOf[address(this)]` to determine how many LP tokens to redeem. Using live balances (not cached reserves) for the redemption amounts is consistent with the "balance-as-truth" design.

## 8. Fee Accounting System

### 8.1 LP Fee (0.30%)

The 0.30% fee is never explicitly computed or transferred. It is implicitly retained in the pool by the invariant check:

```
balance0Adjusted + balance1Adjusted ≥ reserve0 + reserve1 + 1,000,000
```

Because the fee-adjusted balances are strictly less than the actual balances, the pool's actual `k` (`balance0 × balance1`) grows over time as trades execute. This growth in `k` is the LP fee accumulation. LPs capture this value when they burn their shares for a pro-rata portion of the grown reserves.

### 8.2 Protocol Fee (0.05%) — `_mintFee`

When `feeTo != address(0)`, the protocol fee is collected by minting new LP tokens to `feeTo` whenever liquidity is added or removed. The mechanism is based on the growth of `sqrt(k)` since the last liquidity event:

```

function _mintFee(uint128 _reserve0, uint128 _reserve1) private returns (bool feeOn) {
    address feeTo = IUniswapV2Factory(factory).feeTo();
    feeOn = feeTo != address(0);
    uint _kLast = kLast;
    if (feeOn) {
        if (_kLast != 0) {
            uint rootK      = Math.sqrt(uint(_reserve0).mul(_reserve1));
            uint rootKLast = Math.sqrt(_kLast);
            if (rootK > rootKLast) {
                uint numerator  = totalSupply.mul(rootK.sub(rootKLast));
                uint denominator = rootK.mul(5).add(rootKLast);
                uint liquidity  = numerator / denominator;
                if (liquidity > 0) _mint(feeTo, liquidity);
            }
        }
    } else if (_kLast != 0) {
        kLast = 0; // clear kLast when fee is off (saves gas on future swaps)
    }
}

```

The formula for LP shares to mint to the protocol:

$$sm = (\sqrt{k_2} - \sqrt{k_1}) / (5 \cdot \sqrt{k_2} + \sqrt{k_1}) \cdot s_1$$

This formula derives from the whitepaper, where  $\phi = 1/6$ :

$$sm / (sm + s_1) = \phi \cdot f_{1,2} \quad \text{where } f_{1,2} = 1 - \sqrt{k_1}/\sqrt{k_2}$$

Solving for `sm`:

$$\begin{aligned} s_m &= (\sqrt{k_2} - \sqrt{k_1}) / ((1/\phi - 1) \cdot \sqrt{k_2} + \sqrt{k_1}) \cdot s_1 \\ &= (\sqrt{k_2} - \sqrt{k_1}) / (5 \cdot \sqrt{k_2} + \sqrt{k_1}) \cdot s_1 \quad [\text{since } 1/\phi - 1 = 5] \end{aligned}$$

This gives the protocol exactly 1/6 of accumulated LP fees (0.05% of volume), while LPs retain 5/6 (0.25% of volume). Traders always pay exactly 0.30%.

**Gas optimization:** `kLast` is only written when a liquidity event occurs (`mint()` or `burn()`), not on every swap. This keeps the hot swap path cheaper. If the fee is off (`feeTo == address(0)`), `kLast` is zeroed to avoid paying for the `sqrt` computation on every liquidity event.

## 9. Price Oracle (TWAP)

### 9.1 Cumulative Price Accumulators

The oracle works by accumulating `price * elapsed_seconds` since contract inception:

```
price0CumulativeLast += (reserve1 / reserve0) * timeElapsed [at each first interaction per block]
price1CumulativeLast += (reserve0 / reserve1) * timeElapsed
```

Both directions are tracked because the arithmetic mean of A/B  $\neq$  1 / (arithmetic mean of B/A).

### 9.2 TWAP Computation

An external consumer computes the time-weighted average price over  $[t_1, t_2]$  as:

```
TWAP(t1→t2) = (price0CumulativeLast(t2) - price0CumulativeLast(t1)) / (t2 - t1)
```

The contract itself stores no historical snapshots. The consumer must:

1. Read and store `price0CumulativeLast` and `blockTimestamp` at  $t_1$ .
2. At  $t_2$ , read the current values.
3. Divide the difference by elapsed seconds to get the average price in UQ112.112 format.
4. Divide by  $2^{112}$  to obtain a regular decimal.

### 9.3 Precision

Prices are stored as UQ112.112 — the ratio `reserve1 / reserve0` scaled by  $2^{112}$ . For reserves around  $10^{18}$  (18-decimal tokens), this gives:

- Integer part: up to  $2^{112} \approx 5.2 \times 10^{33}$  (never overflows for realistic pairs)
- Fractional precision:  $1 / 2^{112} \approx 1.9 \times 10^{-34}$  (negligible rounding error)

The accumulated value in `price0CumulativeLast` (uint256) can hold up to  $\sim 2^{256} / 2^{112} / 2^{32} \approx 2^{112}$  seconds of price accumulation without overflow — well beyond the heat death of the universe.

### 9.4 Manipulation Resistance

The oracle uses cached reserves (the state from the *end of the previous block*), not live balances. An attacker who manipulates price within block N (by trading) cannot affect the oracle reading for block N — only for block N+1. But if block N+1 includes an arbitrage correction, the manipulation is undone. The cost of a sustained manipulation attack scales linearly with the pool's depth and the length of the TWAP window.

### 9.5 Oracle Liveness

The oracle only progresses when `swap()`, `mint()`, `burn()`, or `sync()` is called. For illiquid pools, the TWAP may be stale. Applications should validate `blockTimestampLast` against the current block timestamp and enforce a staleness threshold.

## 10. Flash Swaps

## 10.1 Mechanics

Flash swaps are a generalization of the `swap()` function. When `data.length > 0`, the Pair sends output tokens to `to` and then calls `IUniswapV2Callee(to).uniswapV2Call(sender, amount0out, amount1out, data)` before checking the invariant.

The callee interface:

```
interface IUniswapV2Callee {
    function uniswapV2Call(
        address sender,
        uint amount0,
        uint amount1,
        bytes calldata data
    ) external;
}
```

Within the callback, the callee has received the tokens and can use them for any purpose. It must ensure that by the time the callback returns, the Pair's balances satisfy the fee-adjusted invariant.

## 10.2 Repayment Modes

There are two valid repayment modes:

**Mode A — Swap repayment:** Repay the other token. If 100 `token0` was borrowed, repay enough `token1` so that  $\text{balance0} \times \text{balance1} \geq k$  after fees. This is a standard swap where output was received first.

**Mode B — Same-token repayment:** Return the borrowed token plus a 0.30% fee. If 100 `token0` was borrowed, return at least  $100 \times 1000/997 \approx 100.3009$  `token0`. Effectively a flash loan at the LP fee rate.

In Mode B, both `amount0In` and `amount1In` may be non-zero (if a partial swap also occurs), which is why the invariant check is expressed as a product of two fee-adjusted balances rather than requiring only one side non-zero.

## 10.3 Use Cases

- **Arbitrage:** Borrow `token0` from pool A, swap in pool B for `token1`, repay pool A. Zero upfront capital required.
- **Collateral swap:** Borrow the repayment asset from a pool, unwind a lending position, use proceeds to repay.
- **Self-liquidation:** Borrow the debt asset, repay lending protocol, receive collateral, sell enough to repay the flash swap.

# 11. UniswapV2ERC20 & Permit

## 11.1 LP Token Standard

`UniswapV2ERC20` implements a standard ERC-20 LP token with:

- `name = "Uniswap V2", symbol = "UNI-V2", decimals = 18` (all constants)
- Standard `transfer`, `transferFrom`, `approve` functions
- `DOMAIN_SEPARATOR` set in constructor using `chainid` assembly opcode (EIP-712 domain)

## 11.2 EIP-2612 Permit

```
bytes32 public constant PERMIT_TYPEHASH = keccak256(
    "Permit(address owner,address spender,uint256 value,uint256 nonce,uint256 deadline)"
);

function permit(
    address owner,
    address spender,
    uint value,
    uint deadline,
    uint8 v, bytes32 r, bytes32 s
) external {
    require(deadline >= block.timestamp, 'UniswapV2: EXPIRED');
```

```

bytes32 digest = keccak256(abi.encodePacked(
    '\x19\x01',
    DOMAIN_SEPARATOR,
    keccak256(abi.encode(PERMIT_TYPEHASH, owner, spender, value, nonces[owner]++, deadline))
));
address recoveredAddress = ecrecover(digest, v, r, s);
require(
    recoveredAddress != address(0) && recoveredAddress == owner,
    'UniswapV2: INVALID_SIGNATURE'
);
_approve(owner, spender, value);
}

```

`permit()` allows a user to authorize a transfer via a signature instead of an on-chain `approve()` transaction. This enables single-transaction liquidity removal (`removeLiquidityWithPermit` in the Router) — the user signs the permit off-chain, includes the signature in the remove-liquidity call, and the Router calls `permit()` then `transferFrom()` in one transaction.

`nonces[owner]++` prevents signature replay. `deadline` prevents indefinite validity of signatures. `DOMAIN_SEPARATOR` binds the signature to this specific chain and contract, preventing cross-chain replay.

## 12. Periphery: Library & Router

### 12.1 UniswapV2Library

Pure computation library. No state. All functions are `internal pure`.

#### **sortTokens:**

```

function sortTokens(address tokenA, address tokenB)
    internal pure returns (address token0, address token1)
{
    require(tokenA != tokenB, 'UniswapV2Library: IDENTICAL_ADDRESSES');
    (token0, token1) = tokenA < tokenB ? (tokenA, tokenB) : (tokenB, tokenA);
    require(token0 != address(0), 'UniswapV2Library: ZERO_ADDRESS');
}

```

**pairFor** — deterministic Pair address:

```

function pairFor(address factory, address tokenA, address tokenB)
    internal pure returns (address pair)
{
    (address token0, address token1) = sortTokens(tokenA, tokenB);
    pair = address(uint(keccak256(abi.encodePacked(
        hex'ff',
        factory,
        keccak256(abi.encodePacked(token0, token1)),
        hex'96e8ac4277198ff8b6f785478aa9a39f403cb768dd02cbee326c3e7da348845f'
    ))));
}

```

Note: `pairFor` does NOT verify the pair exists on-chain. It computes the address purely from inputs. This means it can be called with tokens for which no pair exists yet. The Router and integrators must handle `INSUFFICIENT_LIQUIDITY` reverts.

**getAmountOut** — exact-in quote:

```

function getAmountOut(uint amountIn, uint reserveIn, uint reserveOut)
    internal pure returns (uint amountOut)
{
    require(amountIn > 0, 'UniswapV2Library: INSUFFICIENT_INPUT_AMOUNT');
    require(reserveIn > 0 && reserveOut > 0, 'UniswapV2Library: INSUFFICIENT_LIQUIDITY');
}

```

```

    uint amountInWithFee = amountIn.mul(997);
    uint numerator = amountInWithFee.mul(reserveOut);
    uint denominator = reserveIn.mul(1000).add(amountInWithFee);
    amountOut = numerator / denominator;
}

```

**getAmountIn** — exact-out quote:

```

function getAmountIn(uint amountOut, uint reserveIn, uint reserveOut)
internal pure returns (uint amountIn)
{
    require(amountOut > 0, 'UniswapV2Library: INSUFFICIENT_OUTPUT_AMOUNT');
    require(reserveIn > 0 && reserveOut > 0, 'UniswapV2Library: INSUFFICIENT_LIQUIDITY');
    uint numerator = reserveIn.mul(amountOut).mul(1000);
    uint denominator = reserveOut.sub(amountOut).mul(997);
    amountIn = (numerator / denominator).add(1); // ceiling
}

```

## 12.2 Router02 — Swap Flows

**swapExactTokensForTokens (exact-in, multi-hop):**

```

function swapExactTokensForTokens(
    uint amountIn,
    uint amountOutMin,
    address[] calldata path,
    address to,
    uint deadline
) external ensure(deadline) returns (uint[] memory amounts) {
    amounts = UniswapV2Library.getAmountsOut(factory, amountIn, path);
    require(amounts[amounts.length - 1] >= amountOutMin, 'UniswapV2Router: INSUFFICIENT_OUTPUT_AMOUNT');
    TransferHelper.safeTransferFrom(
        path[0], msg.sender,
        UniswapV2Library.pairFor(factory, path[0], path[1]),
        amounts[0]
    );
    _swap(amounts, path, to);
}

```

The `_swap` internal function iterates `path` and for each hop calls `pair.swap()` directing output to the next pair in the path (or to the final recipient):

```

function _swap(uint[] memory amounts, address[] memory path, address _to) internal {
    for (uint i; i < path.length - 1; i++) {
        (address input, address output) = (path[i], path[i + 1]);
        (address token0,) = UniswapV2Library.sortTokens(input, output);
        uint amountOut = amounts[i + 1];
        (uint amount0Out, uint amount1Out) = input == token0
            ? (uint(0), amountOut)
            : (amountOut, uint(0));
        address to = i < path.length - 2
            ? UniswapV2Library.pairFor(factory, output, path[i + 2])
            : _to;
        IUniswapV2Pair(UniswapV2Library.pairFor(factory, input, output))
            .swap(amount0Out, amount1Out, to, new bytes(0));
    }
}

```

This passes intermediate outputs directly between pairs via the `to` parameter, avoiding the Router holding tokens at any intermediate step. This reduces token transfers and associated gas costs.

**swapExactTokensForTokensSupportingFeeOnTransferTokens:**

The fee-on-transfer variant does not pre-compute expected amounts. Instead it transfers input, then reads the actual balance received at each step, adapting to any tax deducted in transit:

```
function _swapSupportingFeeOnTransferTokens(
    address[] memory path,
    address _to
) internal {
    for (uint i; i < path.length - 1; i++) {
        ...
        uint amountInput;
        uint amountOutput;
        {
            // Get actual balance in the pair (post fee-on-transfer)
            (uint reserve0, uint reserve1,) = pair.getReserves();
            (uint reserveInput, uint reserveOutput) = input == token0
                ? (reserve0, reserve1) : (reserve1, reserve0);
            amountInput = IERC20(input).balanceOf(address(pair)).sub(reserveInput);
            amountOutput = UniswapV2Library.getAmountOut(amountInput, reserveInput, reserveOutput);
        }
        ...
    }
}
```

### 12.3 ETH / WETH Handling

The Router holds a reference to a canonical WETH address. ETH-in calls wrap ETH to WETH before routing:

```
function swapExactETHForTokens(uint amountOutMin, address[] calldata path, ...)
    external payable ensure(deadline)
{
    require(path[0] == WETH, 'UniswapV2Router: INVALID_PATH');
    ...
    IWETH(WETH).deposit{value: amounts[0]}(); // wrap ETH
    assert(IWETH(WETH).transfer(
        UniswapV2Library.pairFor(factory, path[0], path[1]), amounts[0]
    ));
    _swap(amounts, path, to);
}
```

ETH-out calls unwrap WETH at the end:

```
IWETH(WETH).withdraw(amounts[amounts.length - 1]);
TransferHelper.safeTransferETH(to, amounts[amounts.length - 1]);
```

## 13. Non-standard Token Handling

### 13.1 Tokens Without Return Values (USDT, BNB)

Early ERC-20 implementations (USDT, BNB, others) do not return a boolean from `transfer()`. The standard ABI decoder would revert treating the missing return as `false`.

Uniswap v2 handles this in `_safeTransfer()`:

```
require(
    success && (data.length == 0 || abi.decode(data, (bool))),
    'UniswapV2: TRANSFER_FAILED'
);
```

`data.length == 0` accepts no-return tokens as successful (consistent with their intent). This was a deliberate change from v1 which interpreted missing returns as failures.

### 13.2 Fee-on-Transfer Tokens

Tokens that deduct a tax during `transfer()` cause the Pair to receive less than expected. This breaks the standard `getAmountOut` prediction (which assumes full delivery) but does **not** break the core invariant check, because the Pair infers `amountIn` from balance deltas rather than trusting the claimed input.

The consequence is:

- Core `swap()` executes correctly (invariant holds)
- Router `swapExactTokensForTokens` may revert (received less than `amountOutMin`)
- Router `swapExactTokensForTokensSupportingFeeOnTransferTokens` handles this correctly by measuring actual deltas

### 13.3 Reentrant Tokens (ERC-777)

ERC-777 tokens can trigger callbacks via `tokensToSend` / `tokensReceived` hooks during `transfer`. Without protection, a hook could reenter `swap()` or `burn()` mid-execution, allowing fee bypass or double-spend attacks.

The `lock` mutex prevents all reentrancy into `swap()`, `mint()`, `burn()`, `skim()`, and `sync()`. Any reentrant call reverts with '`UniswapV2: LOCKED`'.

### 13.4 Tokens with Balance > 2<sup>112</sup>

`_update()` contains:

```
require(balance0 <= uint112(-1) && balance1 <= uint112(-1), 'UniswapV2: OVERFLOW');
```

Tokens with total supply exceeding  $2^{112} - 1 \approx 5.19 \times 10^{33}$  (about 5 quadrillion with 18 decimals) would cause swaps to fail if the pair holds more than this. `skim()` exists as a recovery mechanism for this case.

## 14. Protocol Constants & Bounds

Constant	Value	Source	Meaning
MINIMUM_LIQUIDITY	1000 (10 <sup>3</sup> )	UniswapV2Pair	Permanently locked LP shares on first mint
LP fee	0.30% (3/1000)	UniswapV2Pair.swap()	Always applied; retained in reserves
Protocol fee	0.05% (1/6 of LP fee)	_mintFee()	Optional; off by default
fee multiplier	997 / 1000	Library formulas	Exact-in fee calculation
Max reserve	$2^{112} - 1 \approx 5.19 \times 10^{33}$	_update() overflow check	uint112 ceiling
Timestamp modulus	$2^{32}$	_update()	uint32 overflow; wraps Feb 2106
Oracle precision	$2^{112}$	UQ112x112.Q112	Fixed-point scale factor
Storage slot (packed)	1 slot (256 bits)	Slot 8	<code>reserve0</code> + <code>reserve1</code> + <code>blockTimestampLast</code>
SELECTOR	<code>bytes4(keccak256("transfer(address,uint256)"))</code>	UniswapV2Pair	Safe ERC-20 transfer selector

Constant	Value	Source	Meaning
init code hash	0x96e8ac42...845f	UniswapV2Library.pairFor	CREATE2 deterministic address derivation

## 15. Notable Design Decisions

### 15.1 Balance-as-Truth Design

The Pair never uses `transferFrom()`. Instead, the caller sends tokens to the Pair before calling `mint()`, `burn()`, or `swap()`. The Pair measures received amounts by comparing live balances against cached reserves:

```
amountIn = balance - (reserve - amountOut)
```

This design makes the Pair agnostic to how tokens arrive (direct transfer, meta-transaction, flash repayment, callback), eliminates the need for `approve()` on the core contract, and enables flash swaps by decoupling output from payment timing.

### 15.2 Core/Periphery Separation

The core (`Factory + Pair`) is intentionally minimal and non-upgradeable. All user-protecting logic (slippage bounds, deadlines, multi-hop routing, WETH wrapping, fee-on-transfer handling) lives in the periphery Router, which is upgradeable independently. This means:

- The fund-holding contracts have a tiny attack surface
- UX improvements can ship without touching audited core logic
- Multiple competing routers can coexist (Router01 and Router02 both exist)

### 15.3 Implicit Fee via Invariant Growth

The LP fee is not computed or transferred explicitly. It is embedded in the K-check: because `balance0Adjusted` and `balance1Adjusted` are less than `balance0` and `balance1`, the post-trade `k` ( $balance0 \times balance1$ ) grows relative to the pre-trade `k` ( $reserve0 \times reserve1$ ). Fee accumulation is visible only as reserve growth — collected when LPs exit.

This is gas-efficient (no per-trade SSTORE for fee tracking) but means LPs cannot inspect their "pending fees" on-chain without external simulation.

### 15.4 Protocol Fee via LP Dilution

The protocol fee does not increase the swap fee paid by traders. Instead, when enabled, it dilutes LP shares by minting a small amount of new LP tokens to `feeTo` on every liquidity event. The amount is proportional to `sqrt(k)` growth since `kLast`. This means:

- Traders always pay exactly 0.30%
- LPs get slightly less of the fee (0.25% instead of 0.30%)
- The protocol receives 0.05%
- The protocol fee is collected lazily (only at mint/burn), not on every swap

### 15.5 Deterministic Pool Addresses (CREATE2)

Pair addresses are deterministic given factory address and token pair. This enables:

- Off-chain computation of all pool addresses for a given factory
- Efficient multi-hop routing without on-chain lookups
- Pool address used as a canonical identifier in `getPair[][]`

The salt is `keccak256(token0 ++ token1)` where tokens are address-sorted. The fixed `init code hash` in `pairFor()` is the keccak256 of the Pair creation bytecode — if the Pair contract changes, this hash must be updated in the Library.

### 15.6 Once-Per-Block Oracle

The TWAP oracle only captures price at the end of each block (technically, at the first interaction per block). This makes it impossible to manipulate the oracle by executing multiple trades in a single block, since only the opening price of each block is accumulated. The cost of manipulation is bounded by the depth of the pool and the TWAP window length.

## 15.7 Overflow Intentionality

Several arithmetic operations are intentionally allowed to overflow:

- **Timestamp subtraction:** `blockTimestamp - blockTimestampLast` in `uint32`: wraps correctly across year-2106 rollover
- **Price cumulative accumulators:** `price0CumulativeLast += ...` in `uint256`: TWAP consumers use differences, so wrapping is safe with correct overflow arithmetic

In Solidity 0.5.x (the core version), overflow does not revert by default. This is required for the intentional overflows above. The use of SafeMath throughout protects against *unintentional* overflow.

---