Updated ULP 1.5 / ArbitrageExecutor.huff v1.1 FAQ

Q: Where does the profit land (address/wallet)?

A: Profit (any token0 remaining after repaying the flash loan + fee) initially accumulates within the deployed ArbitrageExecutor.huff contract address.

NEW: A withdrawToken(address token, address recipient) function has been added. The contract owner (the address that deployed the contract) can call this function to transfer the balance of any specified token from the contract to the desired recipient address.

Q: Are all these addresses current (Balancer, Uniswap V3, etc.)?

A:

Balancer Vault: (0xBA1...C9) - Yes, this is the standard V2 Vault address, very likely current for Optimism and other major EVMs.

Uniswap V3 Pools: (pool\_A, pool\_B) - No, these are provided dynamically via userData by the off-chain Rust bot. The bot must use the correct, current pool addresses for the target network.

Function Selectors & Logic: Yes, these correspond to standard ERC20, Balancer V2 flash loans, and Uniswap V3 interfaces.

Q: Is this flexible enough to handle any pair on Uniswap V3?

A: Yes, the contract logic is flexible. It operates based on the token0, token1, pool\_A, pool\_B, and direction (zeroForOne\_swap1) parameters provided in the userData by the Rust bot. It assumes standard ERC20 token behavior.

Q: How is the pair selected for max potential profit?

A: This critical task is handled entirely by the off-chain Rust bot. The Huff contract is purely an executor. The Rust bot must:

Scan numerous pairs and pools across Uniswap V3 (potentially other DEXs).

Calculate potential arbitrage opportunities.

Accurately estimate profitability after deducting Uniswap fees, Balancer flash loan fees, and expected L2 transaction gas costs.

Select the single most profitable opportunity found during its scan cycle.

Construct the precise userData (pool addresses, token addresses, swap direction flag) for the chosen opportunity.

Initiate the transaction by calling flashLoan on the Balancer Vault, targeting this deployed Huff contract.

Q: How is gas efficiency maximized?

A: Huff Language: Inherently low-level, allowing direct EVM opcode manipulation for minimal overhead compared to Solidity.

Callback Amount Capture: Removed an expensive STATICCALL (balanceOf) between swaps by capturing the amount received directly from the Uniswap callback's arguments, saving significant gas per arbitrage.

Minimal Logic: Executes only the necessary steps on-chain: swaps and approval. Profit calculation, opportunity scanning, and decision-making are off-chain.

Optimized Opcodes: Uses efficient patterns like iszero for checks, jumpi for control flow, and careful memory management.

Q: How does this provide a competitive edge?

A: The primary edge is lower gas cost per execution compared to typical Solidity implementations. This allows the bot to either:

Capture smaller profit margin opportunities that might be unprofitable for higher-gas contracts.

Bid a slightly higher gas price (priority fee) for the same transaction cost as competitors, potentially getting included in blocks more reliably during congestion.

The optimized callback handling further enhances this gas advantage.

Q: What about slippage?

A: This contract version does not enforce on-chain slippage protection (e.g., requiring a minimum output amount from the second swap). It performs exact-input swaps.

Responsibility: Slippage protection relies heavily on the Rust bot's pre-transaction simulation. The bot must calculate the expected output considering potential slippage and only execute if the simulated net profit (after gas, fees, and estimated slippage) is positive and meets its threshold.

Optional Enhancement: A check could be added after swap 2 to verify balanceOf(token0) against the expected minimum, but this would cost extra gas (another STATICCALL). The current design prioritizes minimal on-chain gas, assuming the bot handles pre-flight checks.

Contract Compilation with Huff-Neo

Use the Huff-Neo compiler to compile your Huff contracts. For example, to compile FlashExecutor.huff with the FLASH\_LOAN\_404 macro:  
​

.\tools\huff-neo.exe contracts\FlashExecutor.huff --bytecode --alt-main FLASH\_LOAN\_404

Then, convert the output hex to bytes and write it to the build directory:​

$hex = "your\_compiled\_hex\_output"

$bytes = for ($i = 0; $i -lt $hex.Length; $i += 2) { [Convert]::ToByte($hex.Substring($i, 2), 16) }

[IO.File]::WriteAllBytes("build\flash\_executor.bin", $bytes)

Repeat the process for UniV4Swapper.huff, adjusting the macro name as necessary.​