KeepKey Firmware Test Report

Firmware 7.14.1 | 2026-05-23 16:12 | 143 tests: 122 passed, 21 pending

Sections

C. Core - Device Lifecycle

Fundamental device security operations. Every firmware version must pass these tests. A failure here is an absolute release blocker - these protect seed generation, backup, recovery, and access control.

User Flow

WIPE: Erases all keys and settings, returns to factory state
RESET: Generates cryptographic entropy -> BIP-39 mnemonic displayed on OLED only
RECOVERY: Cipher-based entry (scrambled keyboard on OLED) prevents keyloggers
PIN: Randomized grid on OLED, user enters position not digit
PASSPHRASE: Additional BIP-39 word, empty string = default wallet

Tests: 31

✔C1 test_wipe_device

Wipe device (test_msg_wipedevice.py)
Erases all keys, PIN, settings. Device shows "WIPE DEVICE - Do you want to erase your private keys and settings?" on OLED. User must press button to confirm. After wipe, device is uninitialized - no operations work until a new seed is loaded or generated.

✔C2 test_reset_device

Generate new seed (test_msg_resetdevice.py)
Device generates 256 bits of entropy from hardware RNG, converts to BIP-39 mnemonic, and displays words on OLED one page at a time. Words are NEVER sent to the host. User writes them down as their backup.
OLED needed: Seed word display

✔C3 test_reset_device_pin

Generate seed with PIN (test_msg_resetdevice.py)
Same as C2 but also sets a PIN. PIN is entered twice for confirmation via the randomized 3x3 grid on OLED. Verifies PIN is stored and required for subsequent operations.

✔C4 test_failed_pin

PIN mismatch rejects setup (test_msg_resetdevice.py)
If the user enters different PINs during confirmation, the device rejects the setup. This prevents accidentally setting a PIN the user cannot reproduce.
OLED needed: PIN mismatch warning

✔C5 test_already_initialized

Reject reset on initialized device (test_msg_resetdevice.py)
An already-initialized device must refuse reset without a wipe first. Prevents accidental seed replacement which would strand funds on the old seed.

✔C6 test_load_device_1

Load 12-word mnemonic (debug) (test_msg_loaddevice.py)
Debug-only operation: loads a known 12-word mnemonic for testing. In production, seeds can only be generated on-device or recovered via cipher entry.

✔C7 test_load_device_2

Load 18-word mnemonic (debug) (test_msg_loaddevice.py)
Tests 18-word BIP-39 mnemonic support (192 bits of entropy).

✔C8 test_load_device_3

Load 24-word mnemonic (debug) (test_msg_loaddevice.py)
Tests 24-word BIP-39 mnemonic support (256 bits of entropy, maximum security).

✔C9 test_load_device_utf

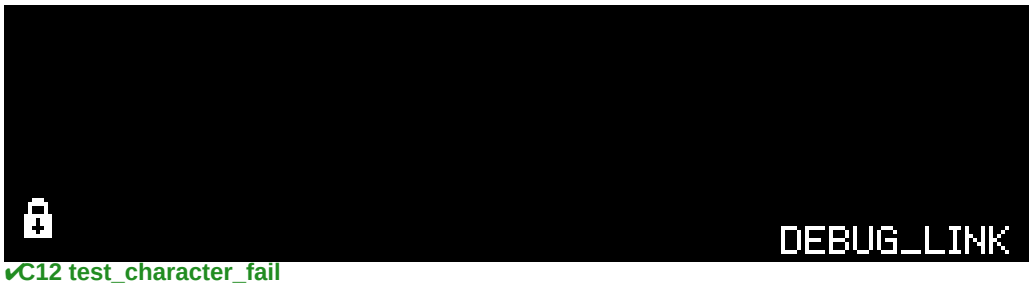
Load with UTF-8 device label (test_msg_loaddevice.py)
Verifies the device handles non-ASCII characters in labels without corruption.

✔C10 test_nopin_nopassphrase

Cipher recovery (no PIN) (test_msg_recoverydevice_cipher.py)
Recovery via scrambled keyboard on OLED. The letter grid is randomized per-character, so even a compromised host cannot determine which letters the user selected. After all words are entered, device verifies BIP-39 checksum and reconstructs the seed.
OLED needed: Cipher grid on OLED

✔C11 test_pin_passphrase

Cipher recovery with PIN + passphrase (test_msg_recoverydevice_cipher.py)
Same recovery flow as C10 but also sets PIN and enables passphrase protection during the recovery process.



✔C12 test_character_fail

Invalid character rejection (test_msg_recoverydevice_cipher.py)
Verifies the cipher entry rejects characters that cannot form any BIP-39 word prefix.

✔C13 test_backspace

Backspace during cipher entry (test_msg_recoverydevice_cipher.py)
User can correct mistakes during word entry without restarting recovery.

✔C14 test_reset_and_recover

Full reset then recover cycle (test_msg_recoverydevice_cipher.py)
End-to-end test: generate seed -> write down words -> wipe -> recover from words -> verify same addresses are derived. Proves the backup/restore cycle works.

✔C15 test_wrong_number_of_words

Wrong word count rejected (test_msg_recoverydevice_cipher.py)
BIP-39 only allows 12, 18, or 24 words. Other counts are rejected immediately.

✔C16 test_correct_same

Dry-run recovery matches (test_msg_recoverydevice_cipher_dryrun.py)
User can verify their backup without wiping the device. Dry-run recovers the seed in memory and compares to the active seed. If they match, user knows their backup is valid.

✔C17 test_correct_notsame

Dry-run detects wrong backup (test_msg_recoverydevice_cipher_dryrun.py)
If the entered words produce a different seed, the device warns the user. This catches transcription errors in the backup before an emergency.

✔C18 test_incorrect

Dry-run rejects bad entry (test_msg_recoverydevice_cipher_dryrun.py)
Invalid words or checksum failure during dry-run are reported to the user.

✔C19 test_set_pin

Set new PIN (test_msg_changepin.py)
Transitions from no-PIN to PIN-protected. The randomized 3x3 grid prevents screen recording attacks - the attacker sees button presses but not which digit they map to.

✔C20 test_change_pin

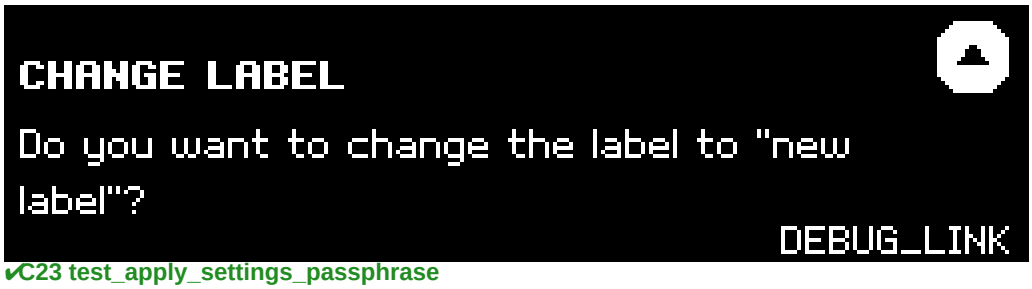
Change existing PIN (test_msg_changepin.py)
Requires entering the current PIN first (proving knowledge), then setting a new one.

✔C21 test_remove_pin

Remove PIN protection (test_msg_changepin.py)
User can disable PIN if physical security is sufficient. Requires current PIN to remove.

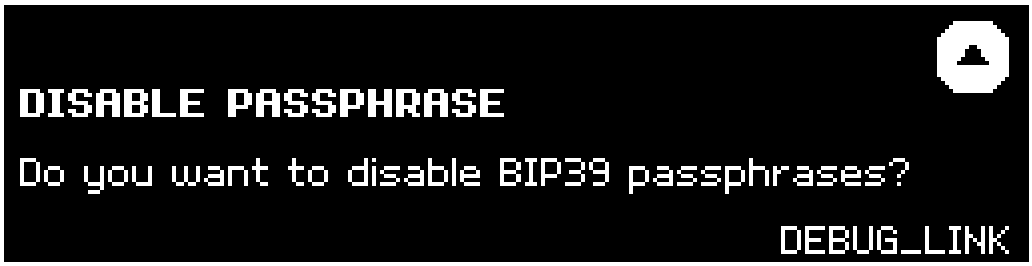
✔C22 test_apply_settings

Change label and language (test_msg_applysettings.py)
Device label appears on OLED during confirmation screens. Helps identify devices when a user has multiple KeepKeys.



✔C23 test_apply_settings_passphrase

Toggle passphrase protection (test_msg_applysettings.py)
Enables/disables BIP-39 passphrase. When enabled, every operation prompts for a passphrase. Different passphrases derive completely different wallets from the same seed.



✔C24 test_clearsession

Clear session state (test_msg_clearsession.py)
Clears cached PIN, passphrase, and session data. Next operation requires re-authentication.

✔C25 test_ping

Ping with button confirmation (test_msg_ping.py)
Basic connectivity test. Verifies the device processes messages and button confirmation works.

✔C26 test_ping_format_specifier_sanitize

Sanitize format specifiers (test_msg_ping.py)
Security test: printf-style format specifiers in ping message must not cause crashes or information leaks. Verifies input sanitization.

✔C27 test_entropy

Hardware RNG entropy (test_msg_getentropy.py)
Reads random bytes from the hardware RNG. Used to verify the entropy source is functional.

✔C28 test_encrypt

Symmetric key encryption (test_msg_cipherkeyvalue.py)
Derives a symmetric key from the HD tree and encrypts data. Used for password manager integrations and encrypted communication.

✔C29 test_decrypt

Symmetric key decryption (test_msg_cipherkeyvalue.py)
Reverse of C28. Verifies encrypt/decrypt round-trips correctly.

✔C30 test_sign

Sign identity challenge (SSH/GPG) (test_msg_signidentity.py)
Signs an identity challenge for SSH login or GPG key derivation. Derives a key from the identity URI and signs the challenge.

-- C31 test_invalid_bip39_word_rejected

BIP-39 invalid word rejected during cipher recovery (test_msg_recoverydevice_cipher.py)
Enter a non-BIP-39 word ("zz") during cipher recovery with enforce_wordlist=True. Firmware must reject immediately with Failure instead of silently accepting.
OLED needed: Wordlist rejection warning

B. Bitcoin

Bitcoin is the primary chain and most extensively tested. Covers legacy P2PKH, P2SH-wrapped SegWit, native SegWit (bech32), and Taproot (P2TR). Transaction signing validates that the device correctly displays every output address and amount, calculates fees, detects change outputs, and resists output substitution attacks. Also covers UTXO forks sharing BTC signing code.

User Flow

ADDRESS: Derive key from BIP-32 path -> display on OLED with QR code -> user verifies against host
SIGN TX: Device shows each output (full address + amount) -> shows fee -> user confirms -> signs
MESSAGE: Show text on OLED -> user confirms -> signs with address-specific key (EIP-191 equivalent)

Tests: 28/28 -- ALL PASSED

✔B1 test_btc

Derive BTC legacy address (test_msg_getaddress.py)
Derives a P2PKH (1...) address from standard BIP-44 path m/44'/0'/0'/0/0. Verifies the address matches the expected value from the test mnemonic.

✔B2 test_ltc

Derive Litecoin address (test_msg_getaddress.py)
LTC uses the same derivation as BTC with coin_type=2. Verifies L... address format.

✔B3 test_tbtc

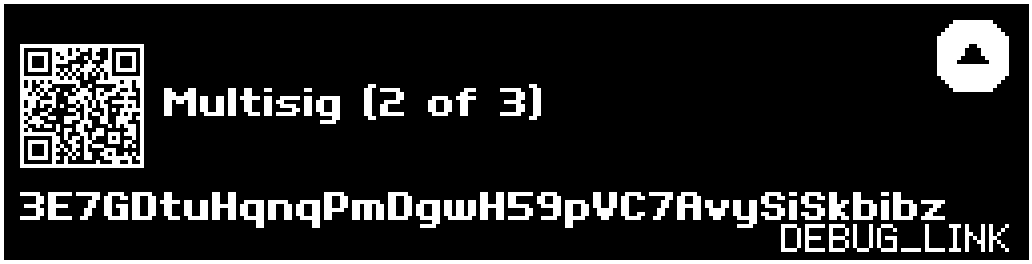
Derive testnet address (test_msg_getaddress.py)
Testnet addresses use different version bytes (m/n prefix). Important for development testing.

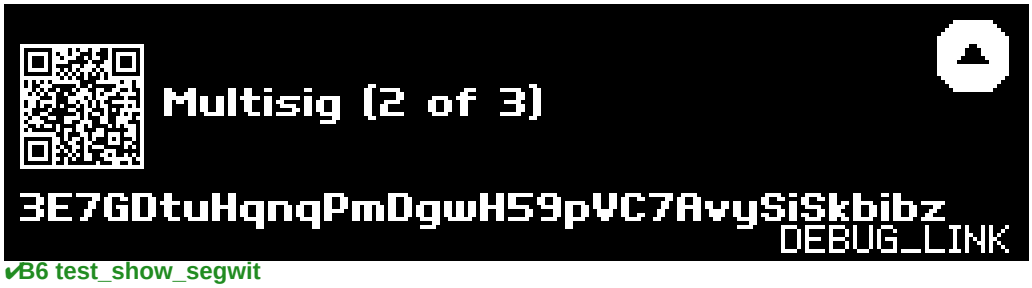
✔B4 test_show

Show BTC address on OLED (test_msg_getaddress_show.py)
Address displayed on OLED with QR code for visual verification. User compares the address shown on the trusted device display against the host application. This is the primary defense against address substitution attacks by compromised hosts.

✔B5 test_show_multisig_3

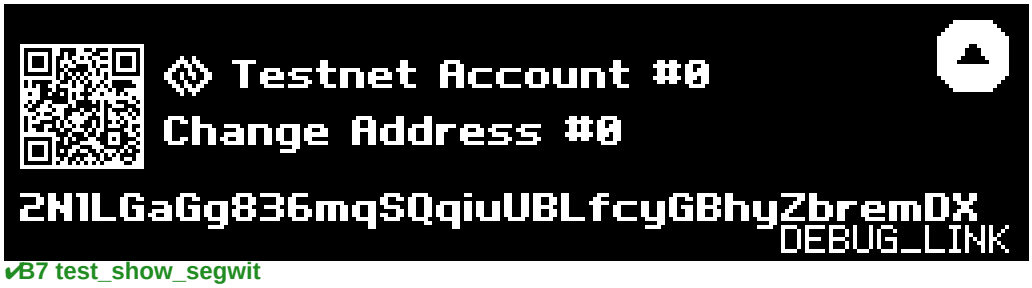
Show 3-of-3 multisig address (test_msg_getaddress_show.py)
Multisig addresses require all co-signer xpubs. Device displays the P2SH multisig address derived from all provided public keys.





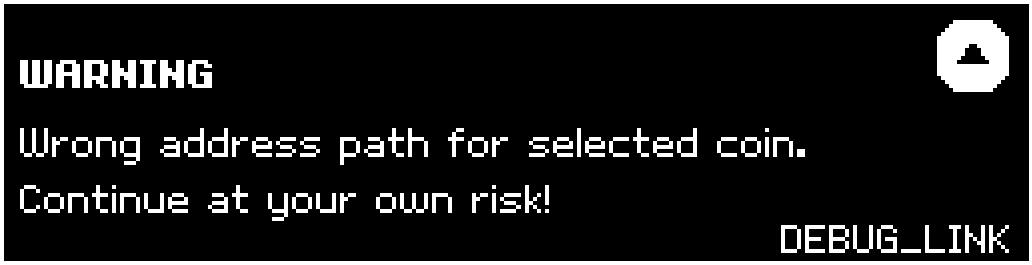
✔B6 test_show_segwit

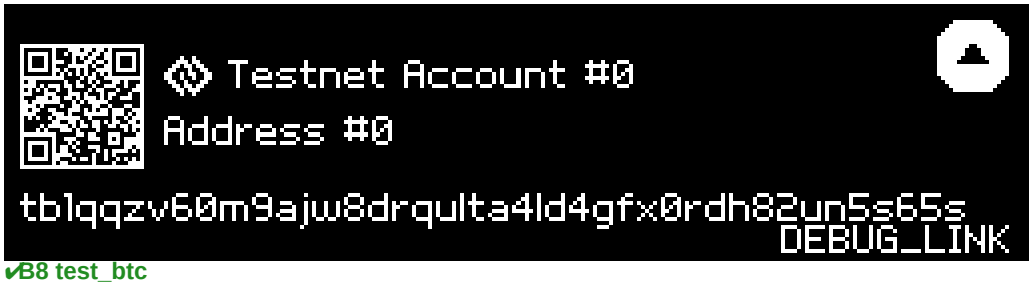
Show SegWit P2SH address (test_msg_getaddress_segwit.py)
P2SH-wrapped SegWit (3... prefix). Backwards compatible with legacy wallets while getting SegWit fee savings.



✔B7 test_show_segwit

Show native SegWit bech32 (test_msg_getaddress_segwit_native.py)
Native SegWit (bc1q... prefix). Lowest fees, modern address format. Verifies bech32 encoding.



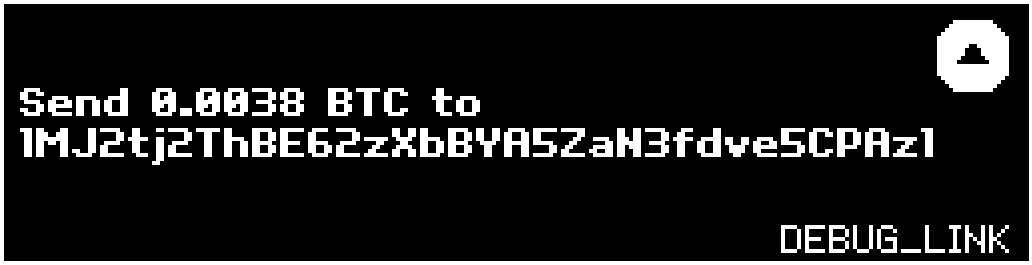


✔B8 test_btc

Get BTC xpub (test_msg_getpublickey.py)
Exports the extended public key for a derivation path. Used by wallet software to derive addresses and monitor balances without the device connected.

✔B9 test_one_one_fee

Sign basic BTC transaction (test_msg_signtx.py)
Simplest case: one input, one output. Device displays "Send X BTC to [address]" with the full recipient address (no truncation), then shows the fee. Verifies the signed transaction is valid.



✔B10 test_one_two_fee

Sign BTC tx with change (test_msg_signtx.py)
One input, two outputs (payment + change). Device must identify the change output (same xpub tree) and only display the payment output to the user.

✔B11 test_two_two

Sign multi-input BTC tx (test_msg_signtx.py)
Two inputs, two outputs. Verifies correct fee calculation across multiple inputs.

✔B12 test_spend_coinbase

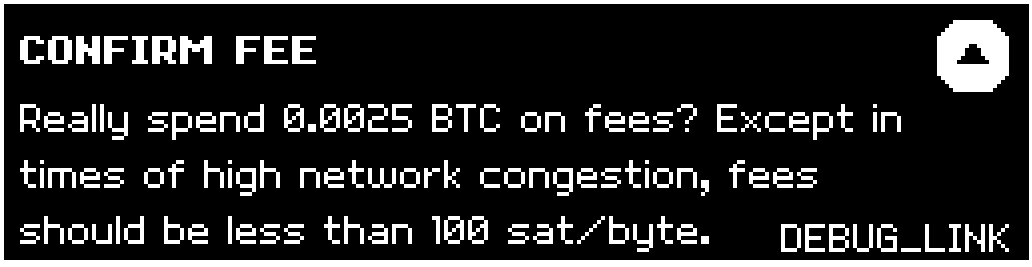
Sign coinbase spend (test_msg_signtx.py)
Spending a coinbase (mining reward) output. Coinbase outputs have special maturity rules.

✔B13 test_lots_of_outputs

Sign tx with many outputs (test_msg_signtx.py)
Stress test with many recipients. Each output is displayed individually on the OLED.

✔B14 test_fee_too_high

Reject excessive fee (test_msg_signtx.py)
If the fee exceeds a safety threshold, the device shows a prominent warning. Protects against fat-finger errors or malicious fee manipulation.



✔B15 test_not_enough_funds

Reject insufficient funds (test_msg_signtx.py)
If inputs don't cover outputs + fee, the device refuses to sign.

✔B16 test_p2sh

Sign P2SH transaction (test_msg_signtx.py)
Pay-to-Script-Hash output. Used for multisig and complex scripts.

✔B17 test_attack_change_outputs

Detect output substitution (test_msg_signtx.py)
Security test: the host attempts to substitute the change output address between the first and second signing pass. Device must detect the mismatch and refuse.

✔B18 test_send_p2sh

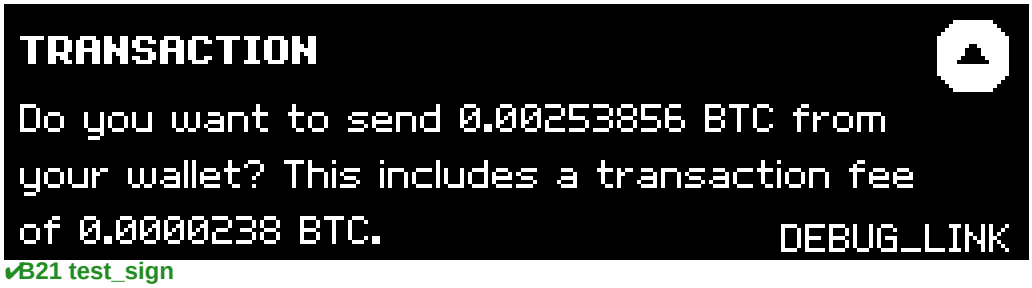
Sign SegWit P2SH tx (test_msg_signtx_segwit.py)
SegWit transaction with P2SH-wrapped inputs. Different signing algorithm (BIP-143).

✔B19 test_send_mixed

Sign mixed legacy+SegWit tx (test_msg_signtx_segwit.py)
Transaction with both legacy and SegWit inputs in the same transaction.

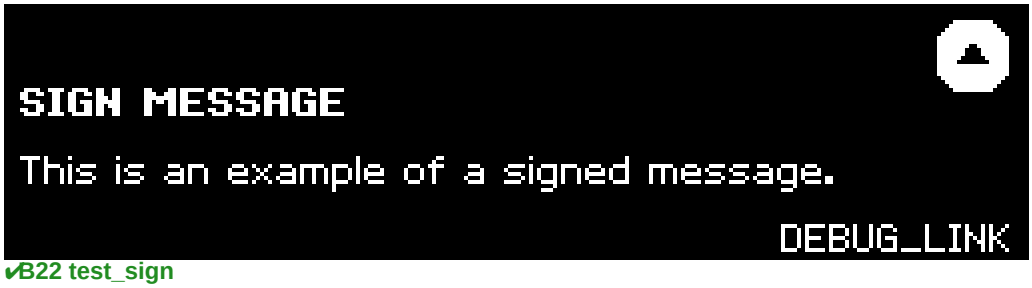
✔B20 test_send_p2tr_only

Sign Taproot P2TR tx (test_msg_signtx_p2tr.py)
Taproot (BIP-341/342) with Schnorr signatures. Newest address type with improved privacy and efficiency.



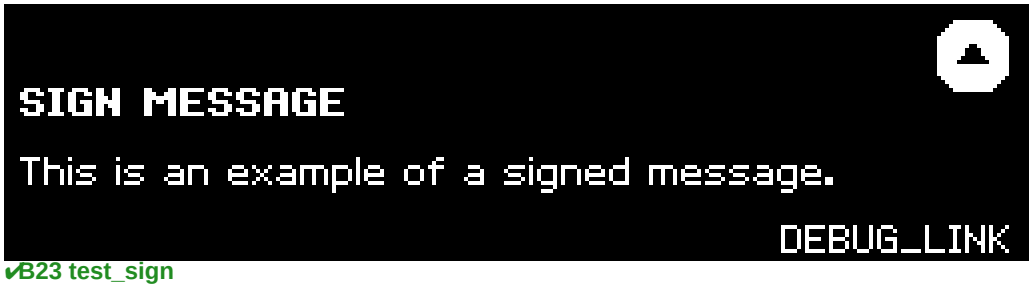
✔B21 test_sign

Sign message with BTC key (test_msg_signmessage.py)
Signs arbitrary text with a BTC address key. Used for proof-of-ownership and login.



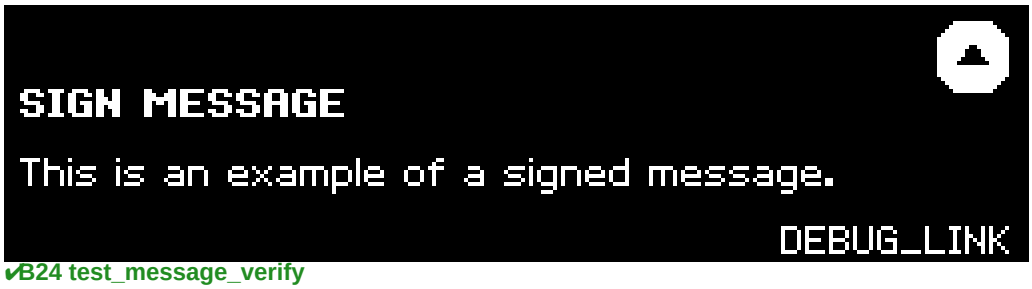
✔B22 test_sign

Sign message with SegWit key (test_msg_signmessage_segwit.py)
Message signing with P2SH-SegWit address key.



✔B23 test_sign

Sign message with bech32 key (test_msg_signmessage_segwit_native.py)
Message signing with native SegWit address key.



✔B24 test_message_verify

Verify signed message (test_msg_verifymessage.py)
Device verifies a message signature against a BTC address.

✔B25 test_send_bitcoin_gold_nochange

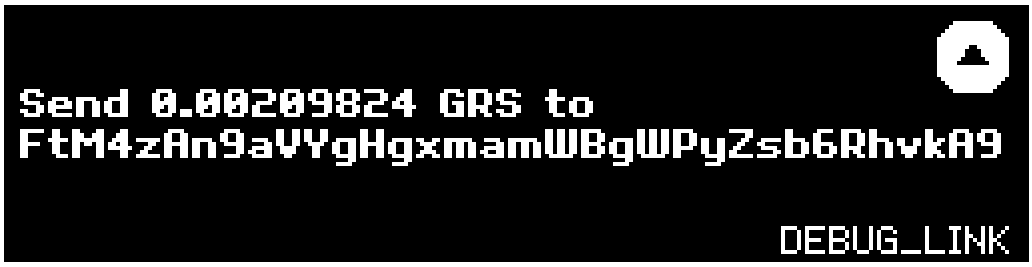
Sign Bitcoin Gold tx (test_msg_signtx_bgold.py)
BTG fork uses same signing code with different chain parameters.

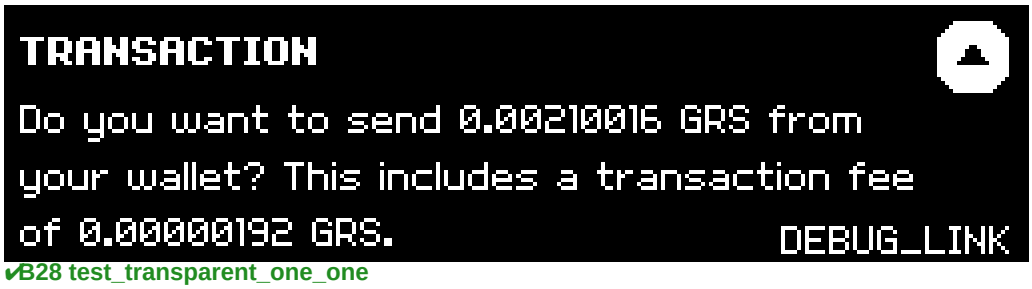
✔B26 test_send_dash

Sign Dash transaction (test_msg_signtx_dash.py)
Dash special transaction types (InstantSend-compatible).

✔B27 test_one_one_fee

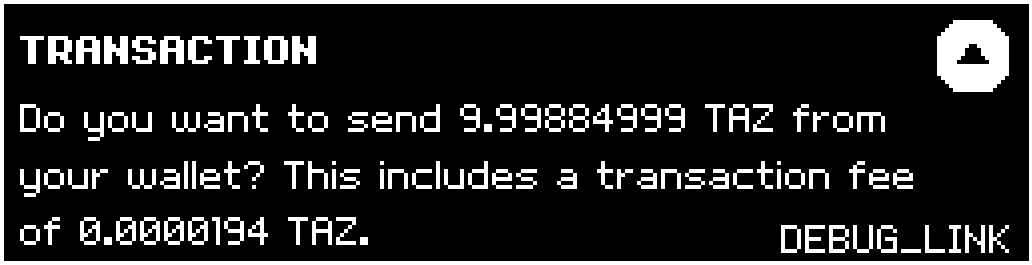
Sign Groestlcoin tx (test_msg_signtx_grs.py)
GRS uses Groestl hash instead of SHA-256d for tx hashing.





✔B28 test_transparent_one_one

Sign Zcash transparent tx (test_msg_signtx_zcash.py)
Zcash transparent transactions use Overwinter/Sapling serialization format with version group IDs and expiry height.



E. Ethereum

Ethereum covers native ETH transfers, ERC-20 tokens, EIP-1559 gas, personal message signing (EIP-191), and contract interactions. The device displays checksummed addresses (EIP-55), values in ETH with 18-decimal precision, and gas parameters.

User Flow

ETH TRANSFER: Show "Send X ETH to 0x..." -> show gas -> confirm -> sign with secp256k1
ERC-20: Decode transfer(to,amount) from contract data -> show token name + amount
EIP-1559: Show maxFeePerGas + maxPriorityFeePerGas (not legacy gasPrice)
MESSAGE: EIP-191 prefix -> show text on OLED -> sign with ETH key

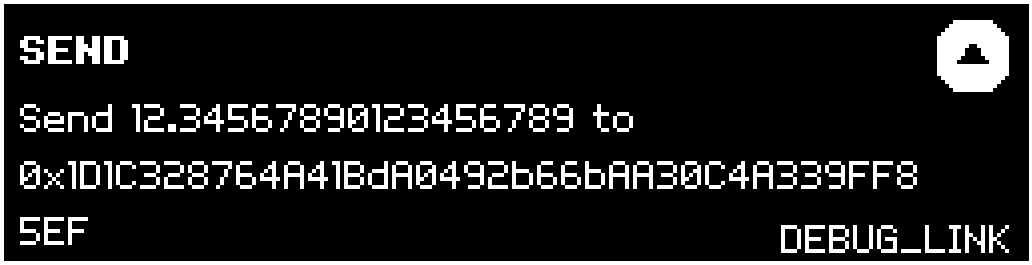
Tests: 16/16 -- ALL PASSED

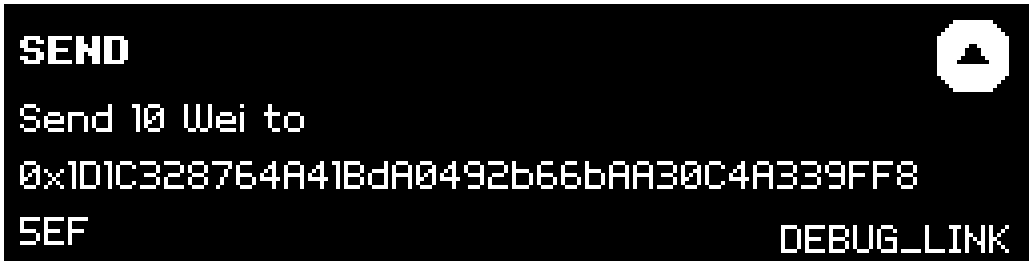
✔E1 test_ethereum_getaddress

Derive ETH address (test_msg_ethereum_getaddress.py)
Standard m/44'/60'/0'/0/0 derivation. EIP-55 checksum address.
OLED needed: ETH address

✔E2 test_ethereum_signtx_nodata

Sign ETH transfer (test_msg_ethereum_signtx.py)
Simple value transfer with no contract data. Device shows recipient + amount + gas.

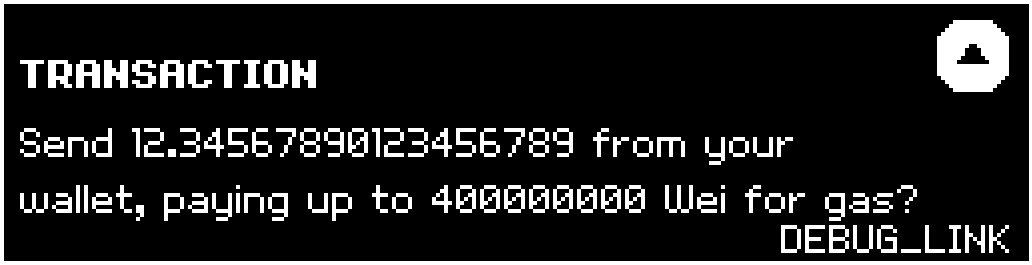


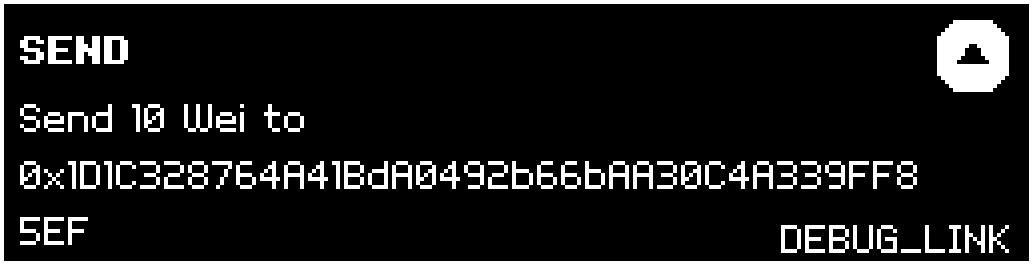


(6 OLED frames captured, showing best 3)

✔E3 test_ethereum_signtx_data

Sign ETH tx with contract data (test_msg_ethereum_signtx.py)
Transaction with data field (contract call). Device shows data as hex since it cannot decode arbitrary ABI without metadata.

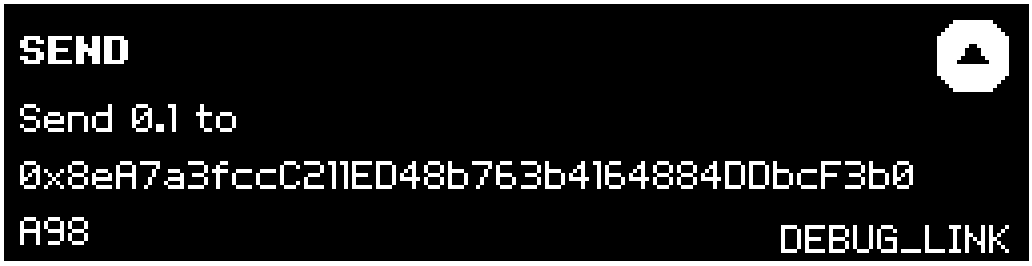


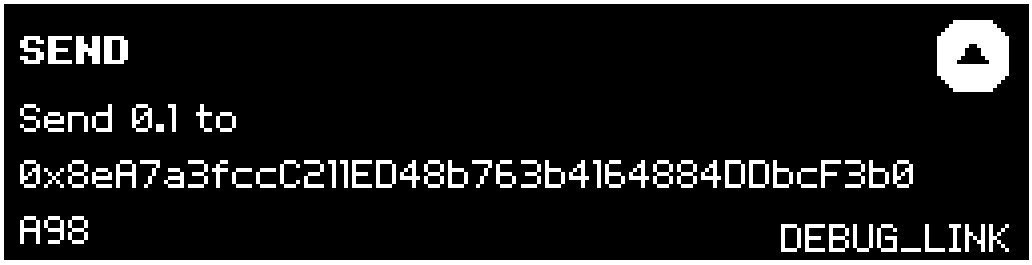


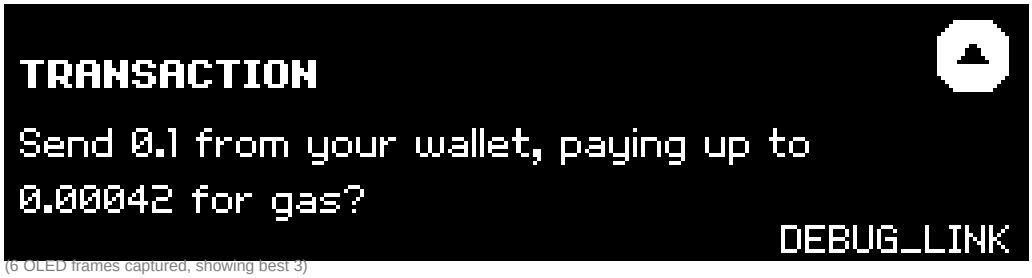
(12 OLED frames captured, showing best 3)

✔E4 test_ethereum_signtx_nodata_eip155

Sign ETH with EIP-155 replay protection (test_msg_ethereum_signtx.py)
Chain ID embedded in signature v value to prevent cross-chain replay attacks.



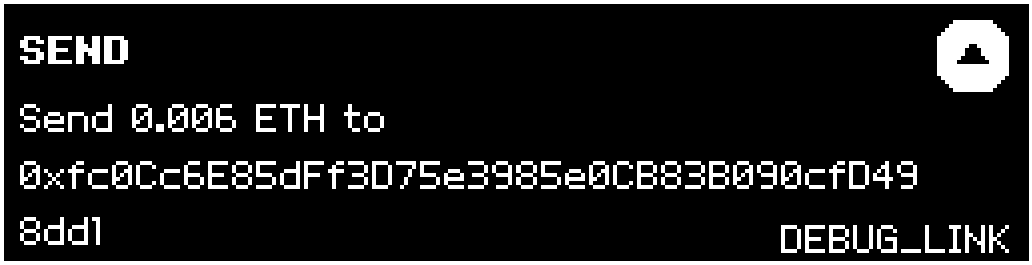




(6 OLED frames captured, showing best 3)

✔E5 test_ethereum_eip_1559

Sign EIP-1559 transaction (test_msg_ethereum_signtx.py)
Type 2 transaction with base fee + priority fee. Device shows both gas parameters.

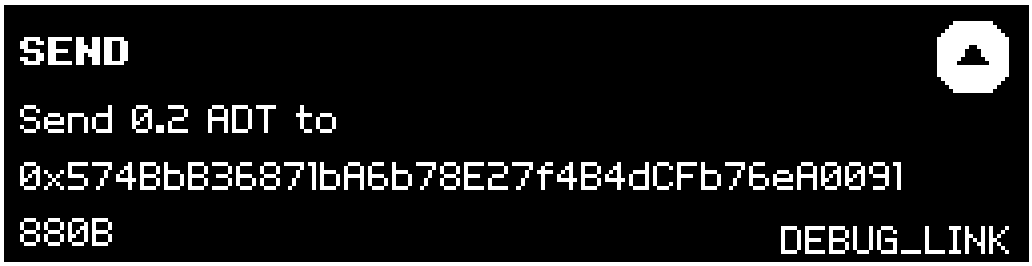


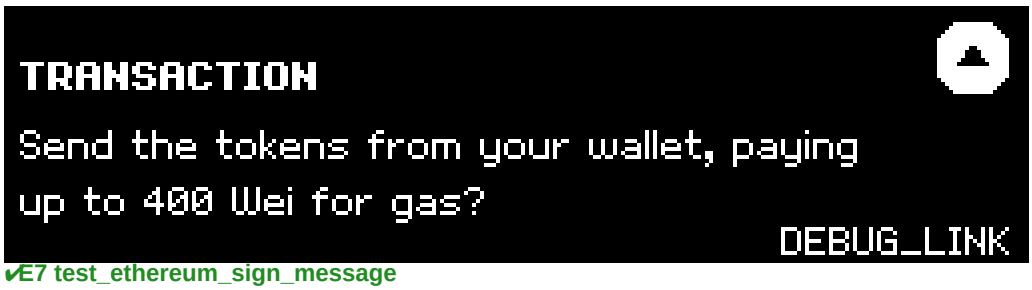
✔E5b test_eip1559_chunked_data_signature_recovers_to_device_address

Sign EIP-1559 with data > 1024 B (chunked transmission) (test_msg_ethereum_signtx_chunked_data_eip1559.py)
Regression for an access-list ordering bug in firmware/ethereum.c ? when data exceeded the 1024-byte single-chunk threshold, the empty access-list byte (0xC0) was hashed between data chunks instead of after them, producing a non-canonical pre-image. The signature recovered to a wrong-but-deterministic address and the broadcast tx was dropped from the mempool. Fixed in 7.14.1.

✔E6 test_ethereum_signtx_knownerc20_eip_1559

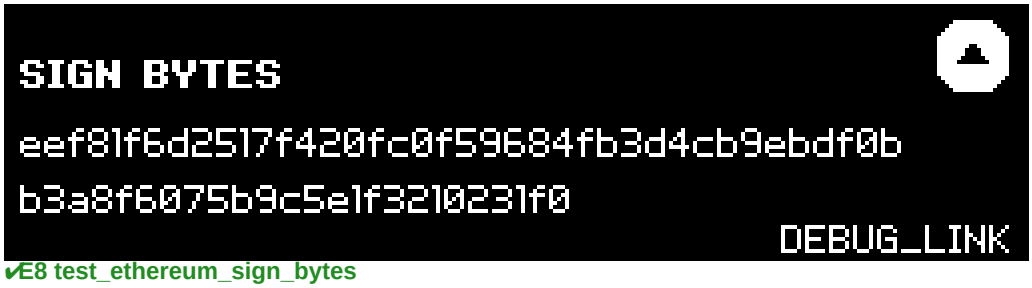
Sign known ERC-20 (EIP-1559) (test_msg_ethereum_signtx.py)
Known token (in firmware token list) via EIP-1559. Shows human-readable token name + amount.





✔E7 test_ethereum_sign_message

Sign personal message (test_msg_ethereum_message.py)
EIP-191 personal_sign. Device shows the message text on OLED for user to verify before signing.



✔E8 test_ethereum_sign_bytes

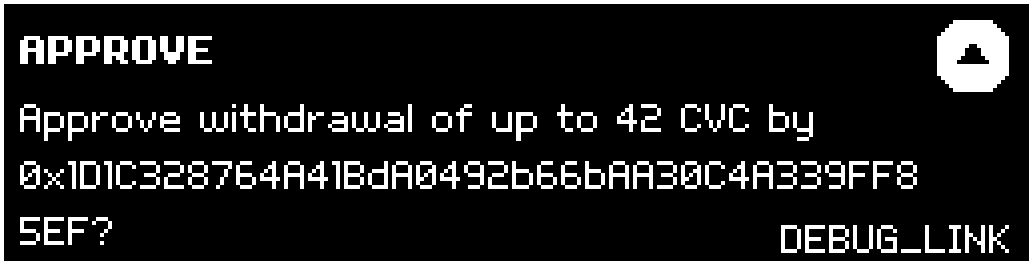
Sign raw bytes (test_msg_ethereum_message.py)
Signs arbitrary bytes (displayed as hex on OLED).

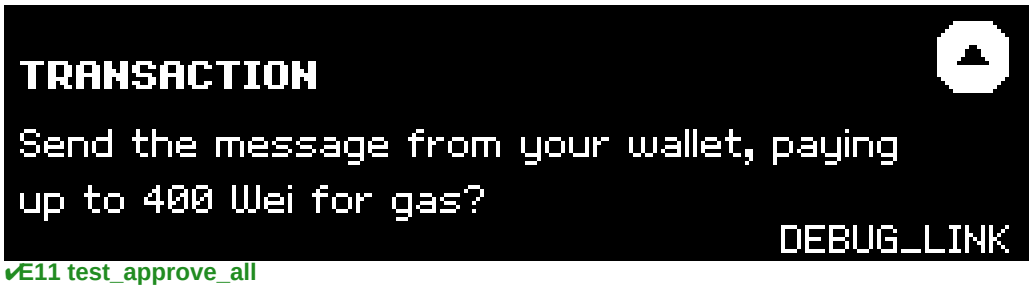
✔E9 test_ethereum_verify_message

Verify ETH signed message (test_msg_ethereum_message.py)
Device-side verification of EIP-191 signed messages.

✔E10 test_approve_some

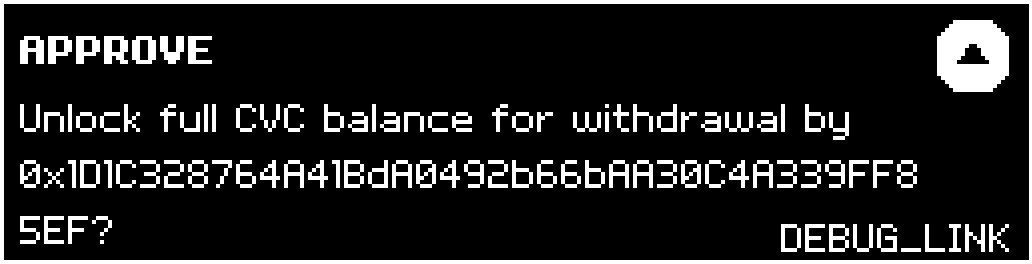
ERC-20 approve specific amount (test_msg_signtx_ethereum_erc20.py)
Token approval for a specific amount. Device shows spender address + approved amount.

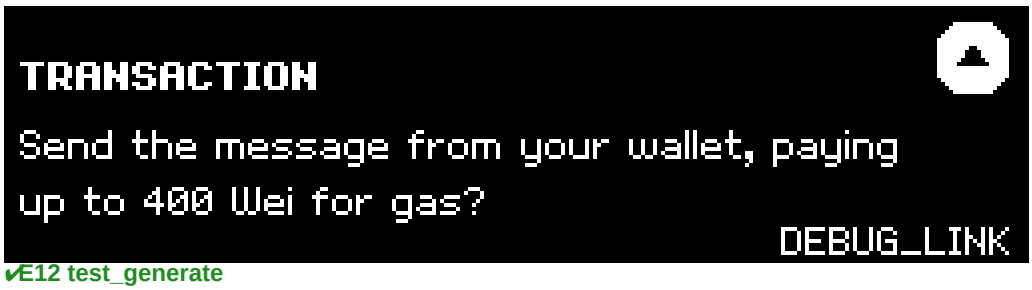




✔E11 test_approve_all

ERC-20 approve unlimited (test_msg_signtx_ethereum_erc20.py)
MAX_UINT256 approval. Device shows "UNLIMITED" warning since this grants infinite spending.





✔E12 test_generate

MakerDAO generate DAI (test_msg_ethereum_makerdao.py)
Complex DeFi contract interaction (MakerDAO CDP).

✔E13 test_sign_salarywithdrawal

Sablier salary withdrawal (test_msg_ethereum_sablier.py)
Streaming payment protocol contract call.

✔E14 test_sign_0x_swap_ETH_to_ERC20

0x swap ETH to ERC-20 (test_msg_ethereum_erc20_0x_signtx.py)
DEX aggregator swap via 0x protocol.

✔E15 test_sign_execTx

Contract function call (test_msg_ethereum_cfunc.py)
Generic contract call signing.

R. Ripple (XRP)

XRP Ledger support for the third-largest cryptocurrency by market cap. XRP uses a unique account-based model (not UTXO) with 20 XRP minimum reserve. Amounts are denominated in drops (1 XRP = 1,000,000 drops). Destination tags are required for exchange deposits to route funds to the correct account. The device displays the full rAddress (34 chars starting with r) and converts drop amounts to human-readable XRP values.

User Flow

ADDRESS: Derive from m/44'/144'/0'/0/0 -> display full rAddress + QR on OLED
SIGN: Host sends Payment tx (destination, amount, fee, destination_tag) -> device shows XRP amount + recipient
FEE: XRP requires a minimum fee (currently 10 drops). Device validates fee is within bounds.

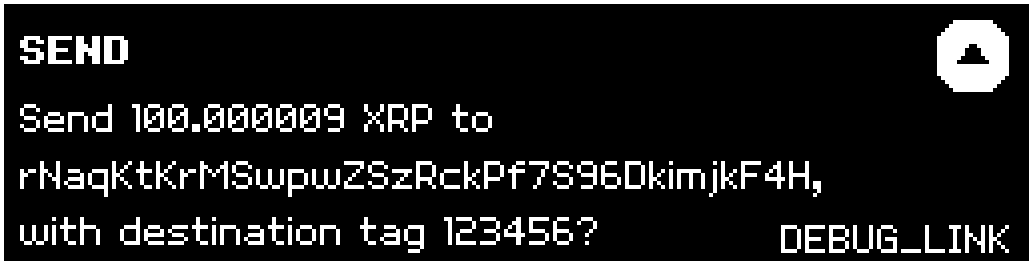
Tests: 3/3 -- ALL PASSED

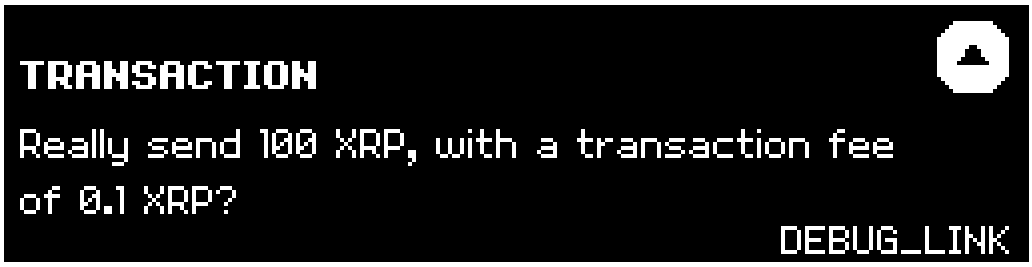
✔R1 test_ripple_get_address

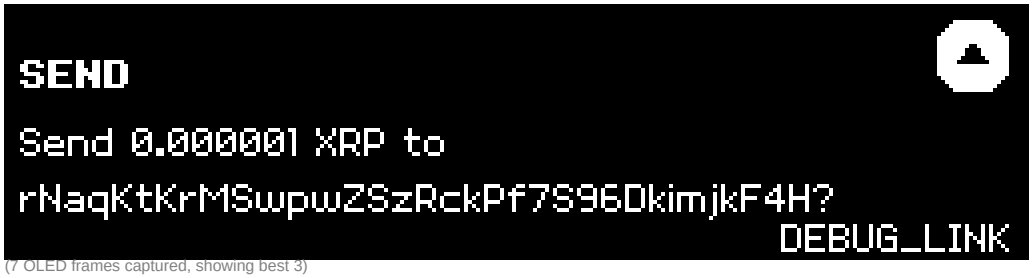
Derive XRP address (test_msg_ripple_get_address.py)
Standard m/44'/144'/0'/0/0 derivation.
OLED needed: XRP address

✔R2 test_sign

Sign XRP payment (test_msg_ripple_sign_tx.py)
Payment with amount in drops (1 XRP = 1,000,000 drops).







(7 OLED frames captured, showing best 3)

✔R3 test_ripple_sign_invalid_fee

Reject invalid fee (test_msg_ripple_sign_tx.py)
Fee outside acceptable range is rejected.

A. Cosmos (ATOM)

Cosmos Hub is the anchor chain for the Cosmos IBC ecosystem. Transactions use amino encoding (legacy Cosmos SDK format). The device supports MsgSend (transfers), MsgDelegate (staking to validators), and MsgWithdrawDelegatorReward (claiming staking rewards). Addresses use bech32 encoding with the cosmos1 prefix. Memo field is critical for exchange deposits and IBC transfers - the device displays it in full on the OLED for user verification.

User Flow

ADDRESS: Derive from m/44'/118'/0'/0/0 -> display cosmos1... bech32 address
SEND: Show recipient address + ATOM amount + memo on OLED -> user confirms
MEMO: Displayed in full - required for exchange deposits (e.g. numeric account ID)

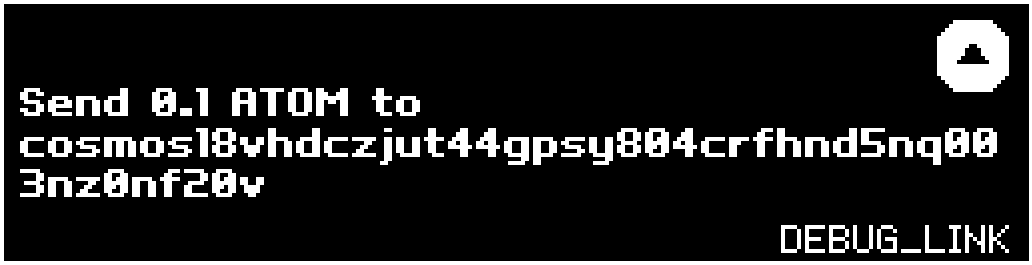
Tests: 3/3 -- ALL PASSED

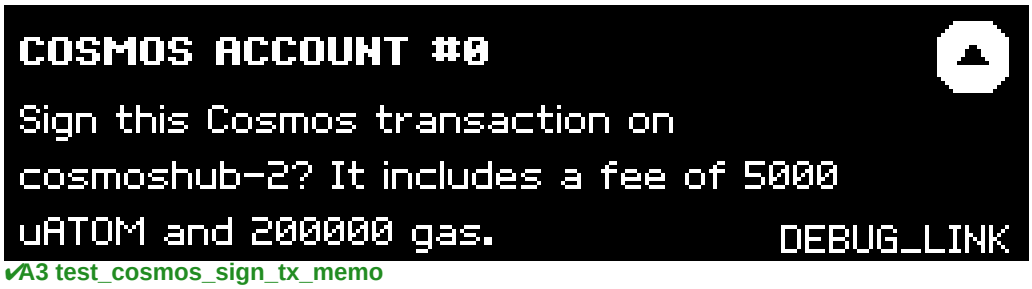
✔A1 test_standard

Derive Cosmos address (test_msg_cosmos_getaddress.py)
Bech32 cosmos1... address from m/44'/118'/0'/0/0.

✔A2 test_cosmos_sign_tx

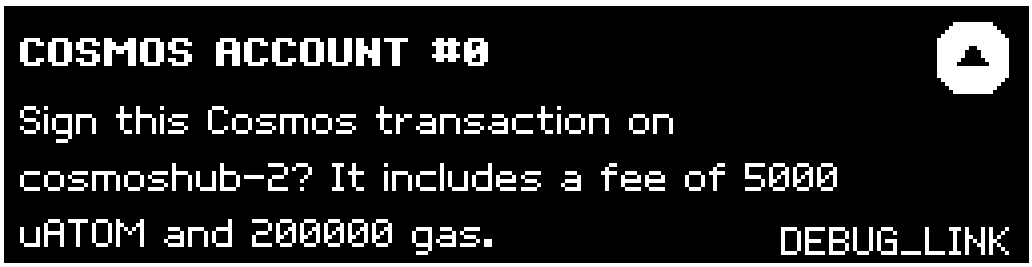
Sign Cosmos send (test_msg_cosmos_signtx.py)
MsgSend with amount + recipient display.





✔A3 test_cosmos_sign_tx_memo

Sign Cosmos with memo (test_msg_cosmos_signtx.py)
Memo field displayed for exchange deposit tags.



H. THORChain

THORChain is a decentralized cross-chain liquidity protocol. Native RUNE transactions use amino encoding with thor1... bech32 addresses. The memo field is the critical security element - it encodes the entire swap/LP instruction (e.g. "SWAP:BTC.BTC:bc1q..." or "=:ETH.ETH:0x..."). A compromised host could substitute the memo destination address to steal funds. The device displays the full memo text on OLED so users can verify the swap destination, pool, and parameters before signing. THORChain also supports LP add/remove operations and deposits.

User Flow

ADDRESS: Derive from m/44'/931'/0'/0/0 -> display thor1... bech32 address
SEND: Show RUNE amount + recipient + full memo text on OLED
SWAP MEMO: "SWAP:BTC.BTC:bc1q..." - user verifies destination chain, asset, and receiving address
LP MEMO: "ADD:BTC.BTC:thor1..." or "WITHDRAW:BTC.BTC:10000" - user verifies pool and basis points

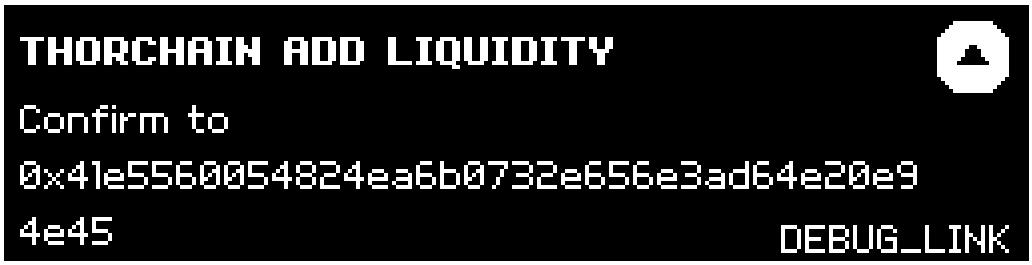
Tests: 4/4 -- ALL PASSED

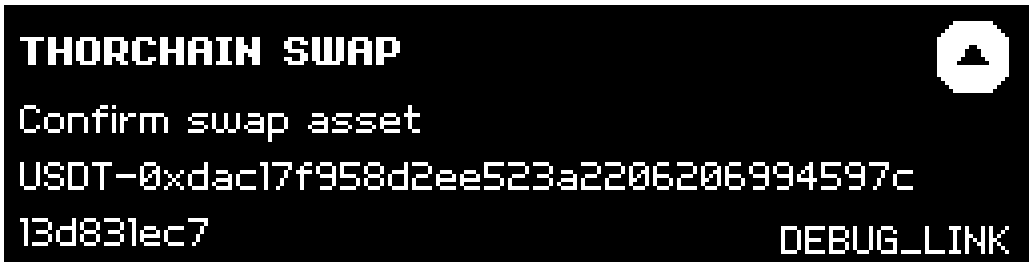
✔H1 test_thorchain_get_address

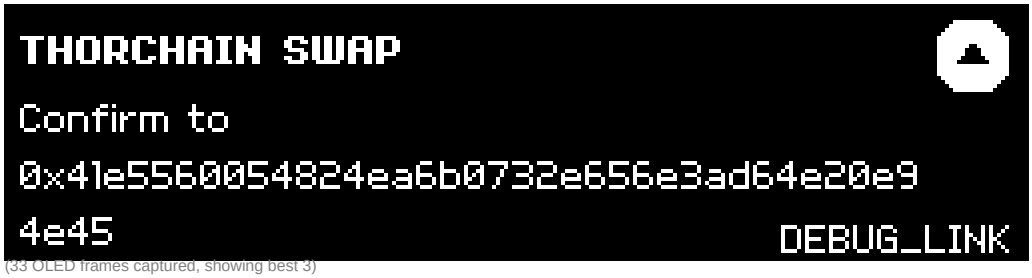
Derive THORChain address (test_msg_thorchain_getaddress.py)
Bech32 thor1... address.

✔H2 test_thorchain_sign_tx

Sign THORChain tx (test_msg_thorchain_signtx.py)
Native RUNE transfer with memo.



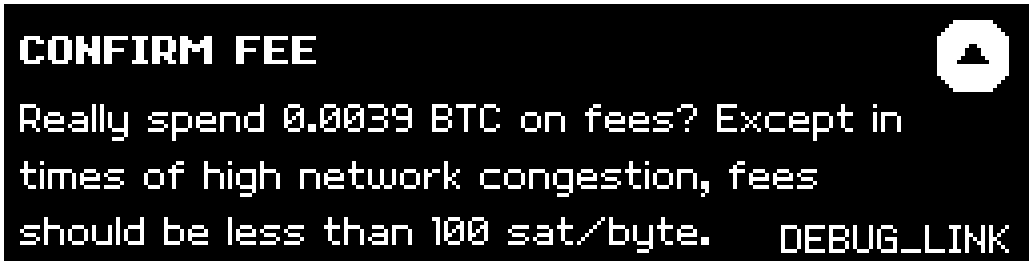


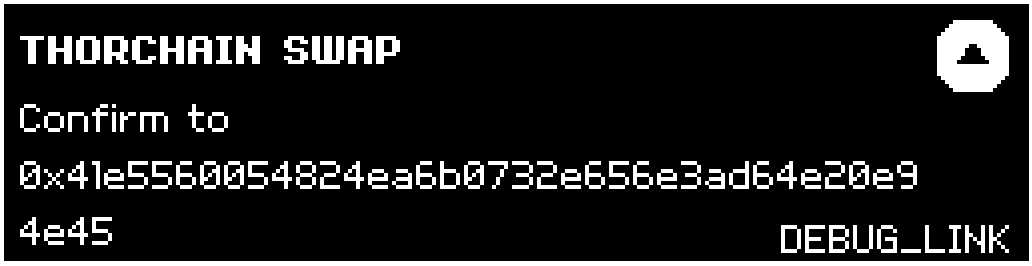


(33 OLED frames captured, showing best 3)

✔H3 test_sign_btc_eth_swap

Sign BTC->ETH swap (test_msg_thorchain_signtx.py)
Cross-chain swap via THORChain memo routing.





(6 OLED frames captured, showing best 3)

✔H4 test_thorchain_sign_tx_deposit

Sign THORChain deposit (test_msg_2thorchain_signtx.py)
LP deposit transaction.

M. Maya Protocol

Maya Protocol is a THORChain fork providing cross-chain liquidity with its native CACAO token. Uses identical amino transaction format and memo-based routing as THORChain but with maya1... bech32 addresses. Maya bridges assets between Bitcoin, Ethereum, THORChain, Dash, and Kujira. The same memo security considerations apply - the device must display the full memo for swap destination verification.

User Flow

ADDRESS: Derive from m/44'/931'/0'/0/0 -> display maya1... bech32 address
SEND: Show CACAO amount + recipient + full memo on OLED
SWAP: Same memo format as THORChain with Maya-specific pool routing

Tests: 3/3 -- ALL PASSED

✔M1 test_mayachain_get_address

Derive Maya address (test_msg_mayachain_getaddress.py)
Bech32 maya1... address.

✔M2 test_sign_btc_eth_swap

Sign BTC-ETH swap via Maya (test_msg_mayachain_signtx.py)
Cross-chain swap via Maya memo routing.

✔M3 test_sign_eth_add_liquidity

Sign swap via Maya (test_msg_mayachain_signtx.py)
Cross-chain swap via Maya memo routing.

O. EOS

EOS chain support with action-based transaction model. Unlike UTXO or account-based chains, EOS transactions contain a list of actions, each targeting a specific smart contract. The device displays each action individually for user review. Covers the core eosio system actions: token transfers, CPU/NET bandwidth delegation, block producer voting, and account authority management (updateauth, linkauth, newaccount). EOS uses a unique account name system (12-char names) instead of addresses.

User Flow

PUBKEY: Derive EOS public key from m/44'/194'/0'/0/0 (EOS format with EOS prefix)
SIGN TX: Host sends action list -> device displays each action with contract + data -> signs
STAKING: delegatebw/undelegatebw for CPU/NET resource management
GOVERNANCE: voteproducer to select block producers

Tests: 4/4 -- ALL PASSED

✔O1 test_trezor

Derive EOS public key (test_msg_eos_getpublickey.py)
EOS public key from m/44'/194'/0'/0/0.

✔O2 test_transfer

Sign EOS transfer (test_msg_eos_signtx.py)
eosio.token::transfer action.

✔O3 test_delegatebw

Delegate bandwidth (test_msg_eos_signtx.py)
CPU/NET resource staking.

✔O4 test_voteproducer

Vote for producer (test_msg_eos_signtx.py)
Block producer voting.

W. Nano

Nano uses a unique block-lattice architecture where each account has its own blockchain. Transactions are feeless and near-instant. The device validates balance encoding for Nano state blocks, which represent the entire account state (balance, representative, link) in a single block. Balance values use 128-bit raw amounts (1 Nano = 10^30 raw).

User Flow

ENCODE: Validate 128-bit balance representation for state block construction
STATE BLOCK: account + previous + representative + balance + link -> hash -> sign

Tests: 1/1 -- ALL PASSED

✔W1 test_encode_balance

Encode Nano balance (test_msg_nano_signtx.py)
Validates the 128-bit balance encoding used in Nano state blocks. Incorrect encoding would cause fund loss or invalid transactions on the block-lattice.

V. EVM Clear-Signing [NEW]

NEW: Verified transaction metadata for EVM contracts. Host sends a signed blob with contract name, function, and decoded parameters. Device verifies blob signature against trusted key, then shows human-readable details with VERIFIED icon. Blind-sign policy gating is deferred to firmware 7.15+.

User Flow

CLEAR-SIGN: Signed metadata -> verify signature -> VERIFIED icon + method + decoded args
BLIND SIGN: No metadata + AdvancedMode on -> contract data signed (no gate until 7.15+)

Tests: 8

-- V1 test_valid_metadata_returns_verified

Valid metadata accepted (test_msg_ethereum_clear_signing.py)
Correctly signed metadata blob is accepted. Device shows VERIFIED icon with decoded method name and contract address.
OLED needed: VERIFIED icon + method

-- V2 test_wrong_key_returns_malformed

Wrong signing key rejected (test_msg_ethereum_clear_signing.py)
Metadata signed with wrong key is rejected as malformed.

-- V3 test_tampered_method_returns_malformed

Tampered method rejected (test_msg_ethereum_clear_signing.py)
Modified method name in blob fails signature check.

-- V4 test_tampered_contract_returns_malformed

Tampered contract rejected (test_msg_ethereum_clear_signing.py)
Modified contract address fails signature check.

-- V5 test_no_metadata_then_sign_unchanged

No metadata = blind sign path (test_msg_ethereum_clear_signing.py)
Without metadata, transaction goes through existing blind-sign path.
OLED needed: Blind sign warning

✔V6 test_signature_verification

Signature verification math (test_msg_ethereum_clear_signing.py)
Unit test for the metadata blob signature algorithm.

✔V7 test_tampered_blob_fails_verification

Tampered blob fails (test_msg_ethereum_clear_signing.py)
Any byte change in the blob invalidates the signature.

✔V8 test_ethereum_blind_sign_allowed

Blind sign permitted (AdvancedMode ON) (test_msg_ethereum_signtx.py)
Contract data with AdvancedMode enabled. Device allows signing. Blind-sign blocking deferred to 7.15+.

S. Solana [NEW]

NEW: Full Solana with Ed25519 (SLIP-10), base58 addresses, 37 instruction types across 7 programs. Key security fix: full 44-character address display replaces old 8-char truncation that was a spoofing vector.

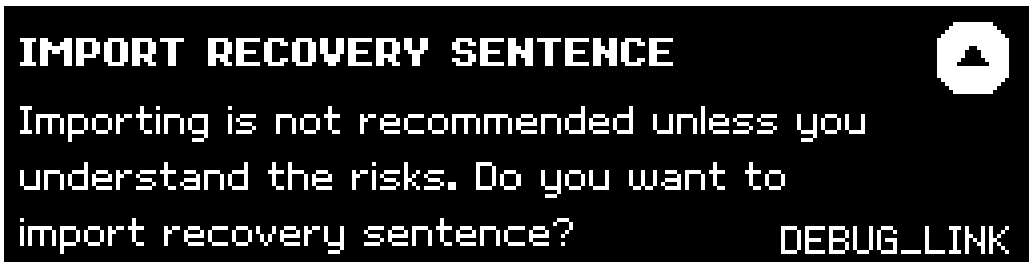
User Flow

ADDRESS: m/44'/501'/0' Ed25519 -> full 44-char base58 on OLED
SIGN TX: Parse instructions -> per-instruction confirmation -> Ed25519 sign
SIGN MESSAGE: Arbitrary bytes -> hex display -> Ed25519 sign

Tests: 13/13 -- ALL PASSED

✔S1 test_solana_get_address

Derive Solana address (test_msg_solana_getaddress.py)
Full 44-character base58 address displayed on OLED.



✔S2 test_solana_different_accounts

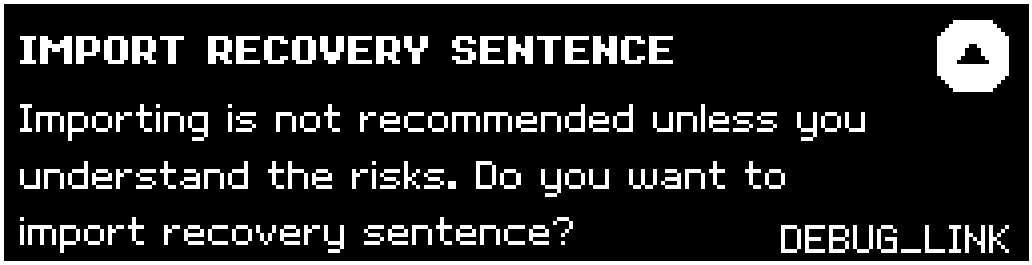
Different account indices (test_msg_solana_getaddress.py)
Verifies different accounts produce different addresses.

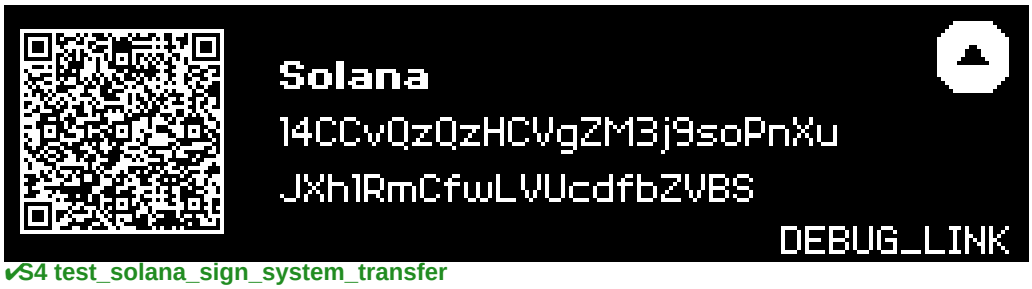
✔S3 test_solana_deterministic

Deterministic derivation (test_msg_solana_getaddress.py)
Same path always produces same address.

✔S3b test_solana_show_address

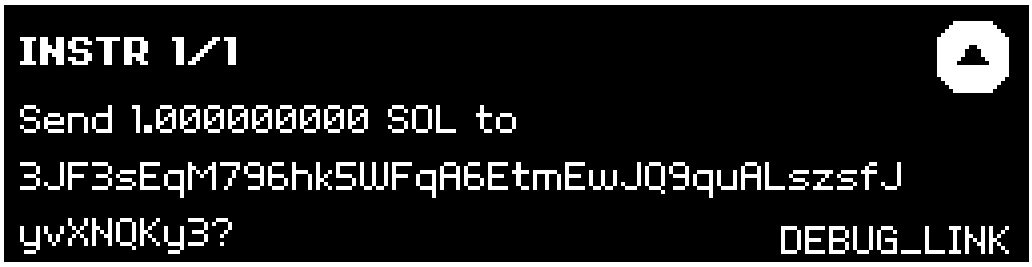
Show address on OLED (test_msg_solana_getaddress.py)
Full 44-char base58 address with QR code on OLED display.

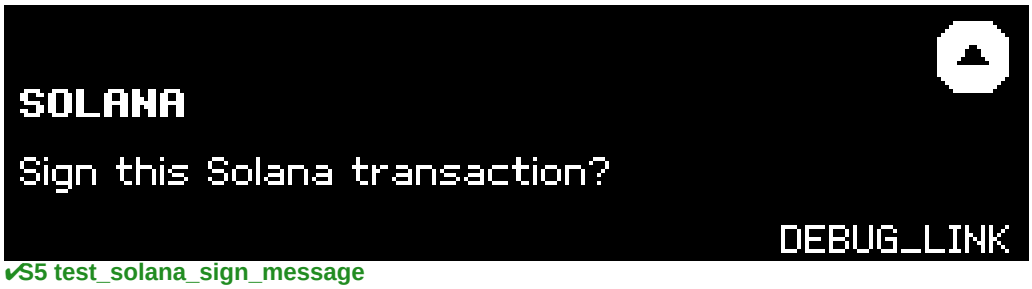




✔S4 test_solana_sign_system_transfer

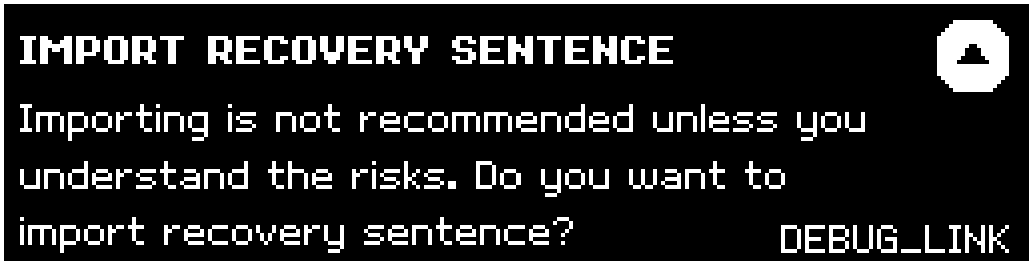
Sign SOL transfer (test_msg_solana_signtx.py)
System::Transfer with full address + amount display.





✔S5 test_solana_sign_message

Sign Solana message (test_msg_solana_signtx.py)
Arbitrary message signing with Ed25519 key. Requires AdvancedMode policy (no domain separation).



(6 OLED frames captured, showing best 3)

✔S6 test_solana_sign_empty_rejected

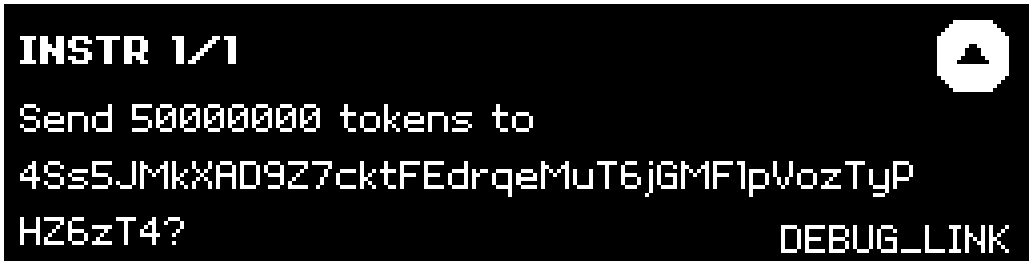
Empty tx rejected (test_msg_solana_signtx.py)
Zero-length transaction data is refused.

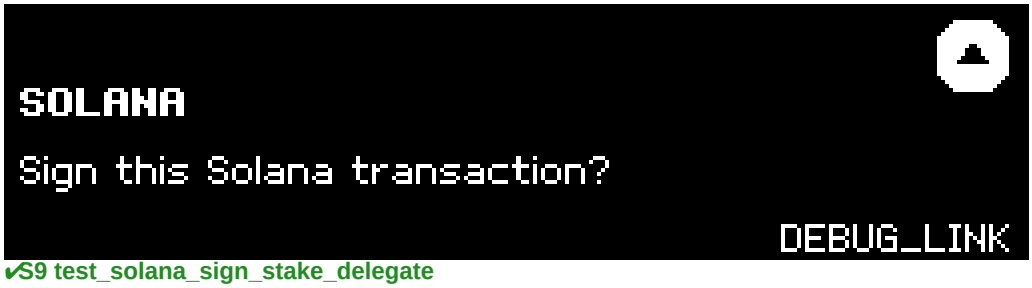
✔S7 test_solana_sign_deterministic

Deterministic signing (test_msg_solana_signtx.py)
Same tx always produces same signature.

✔S8 test_solana_sign_token_transfer

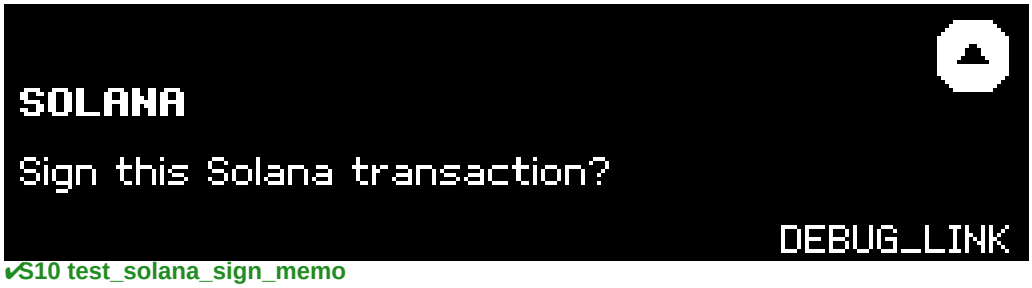
SPL Token transfer (test_msg_solana_signtx.py)
Send SPL tokens to destination. OLED shows token amount and recipient address.





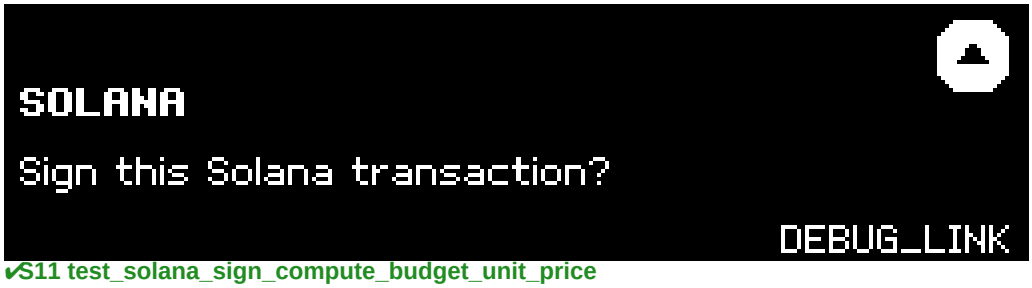
✔S9 test_solana_sign_stake_delegate

Stake delegate (test_msg_solana_signtx.py)
Delegate SOL to a validator for staking rewards. OLED shows delegate confirmation.



✔S10 test_solana_sign_memo

Memo instruction (test_msg_solana_signtx.py)
Attach memo text to transaction. OLED shows memo content.

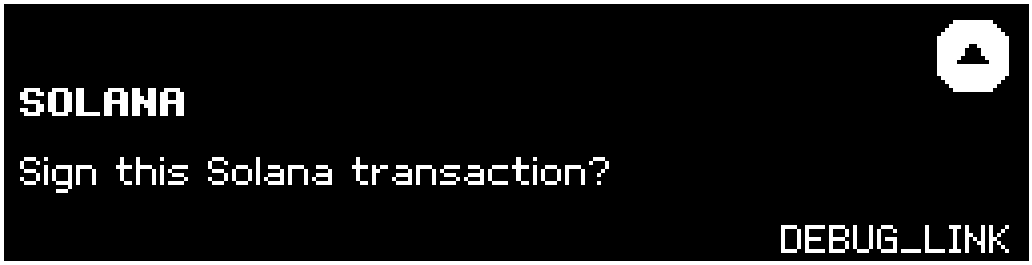


✔S11 test_solana_sign_compute_budget_unit_price

Compute budget unit price (test_msg_solana_signtx.py)
Set priority fee for transaction. OLED shows compute unit price.

✔S12 test_solana_sign_token_transfer_with_metadata

SPL Token with metadata (test_msg_solana_signtx.py)
Token transfer with SolanaTokenInfo (mint, symbol, decimals). OLED shows human-readable token name.



T. TRON [NEW]

NEW: TRON with secp256k1 signing, base58 addresses. Blind-sign via raw_data. Structured reconstruct-then-sign and TRC-20 clear-signing deferred to 7.15+.

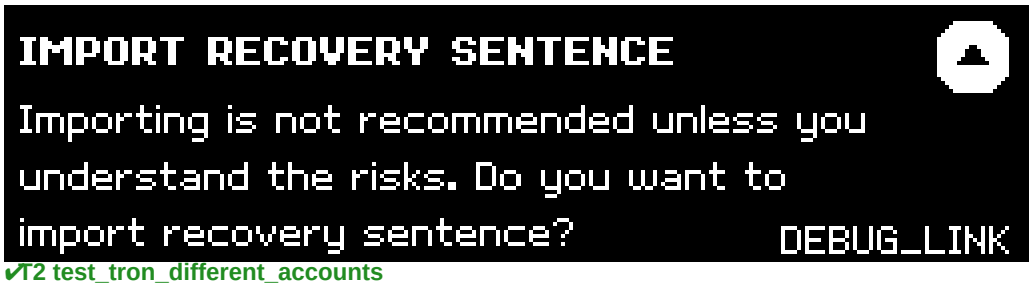
User Flow

ADDRESS: m/44'/195'/0'/0/0 -> full 34-char base58 TRON address
BLIND-SIGN: Raw protobuf data -> hash + sign

Tests: 6/6 -- ALL PASSED

✔T1 test_tron_get_address

Derive TRON address (test_msg_tron_getaddress.py)
Full 34-character base58 address.



✔T2 test_tron_different_accounts

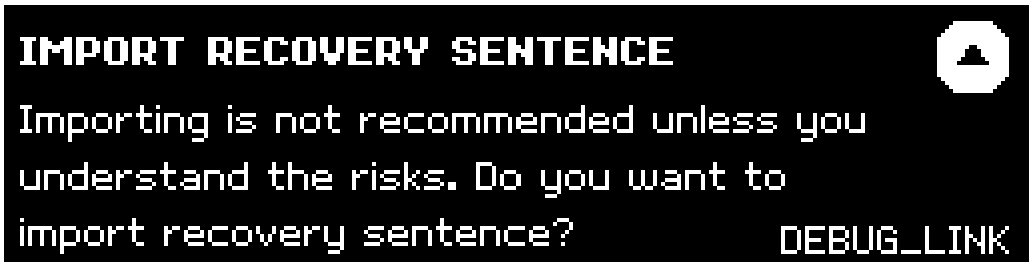
Different accounts (test_msg_tron_getaddress.py)
Different indices produce different addresses.

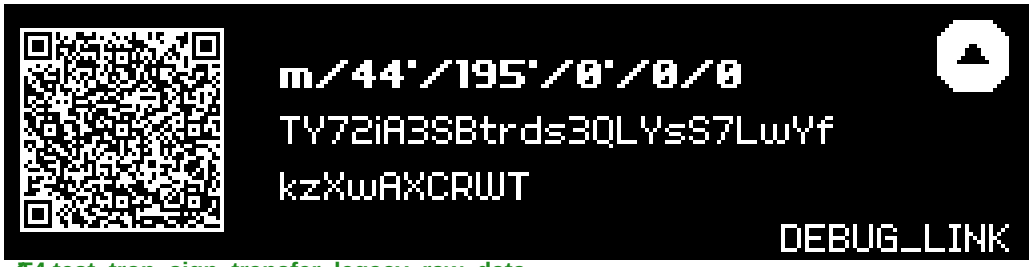
✔T3 test_tron_deterministic

Deterministic derivation (test_msg_tron_getaddress.py)
Same path always produces same address.

✔T3b test_tron_show_address

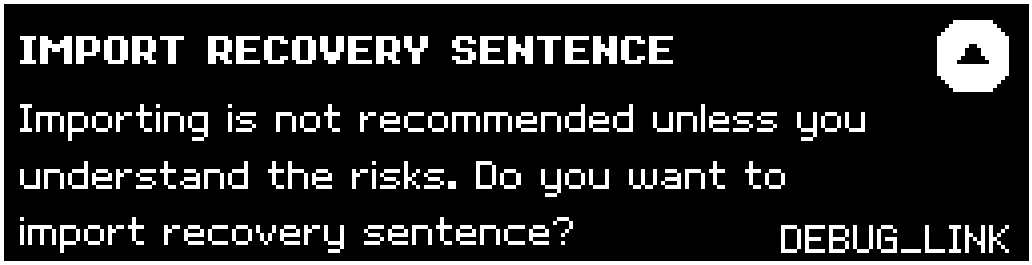
Show address on OLED (test_msg_tron_getaddress.py)
Full 34-char Base58Check TRON address with QR code.

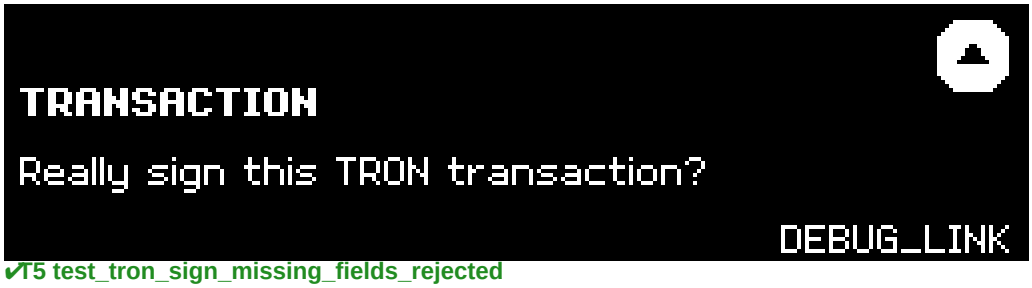




✔T4 test_tron_sign_transfer_legacy_raw_data

Sign TRX blind (raw_data) (test_msg_tron_signtx.py)
Raw protobuf data triggers blind sign path. Shows amount + address if provided.





✔T5 test_tron_sign_missing_fields_rejected

Missing fields rejected (test_msg_tron_signtx.py)
Incomplete transaction data is refused.

N. TON [NEW]

NEW: TON v4r2 wallet contracts. Ed25519 signing with structured field display. Blind-sign for raw transactions. Memo/comment support. Full clear-sign with cell tree reconstruction deferred to 7.15+.

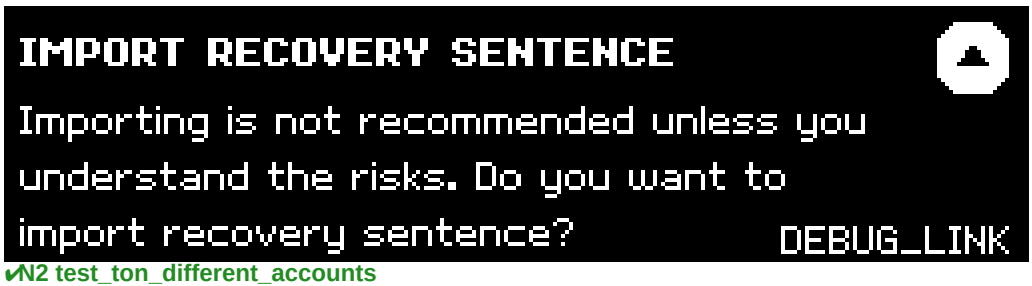
User Flow

ADDRESS: m/44'/607'/0' -> full 48-char base64url TON address
STRUCTURED: Amount + address + memo shown as display context -> sign
BLIND-SIGN: Raw tx without structured fields -> "BLIND SIGNATURE" warning

Tests: 8/8 -- ALL PASSED

✔N1 test_ton_get_address

Derive TON address (test_msg_ton_getaddress.py)
Full 48-character base64url address.

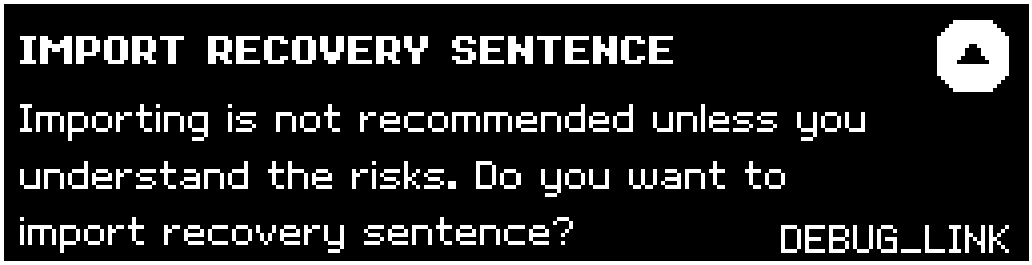


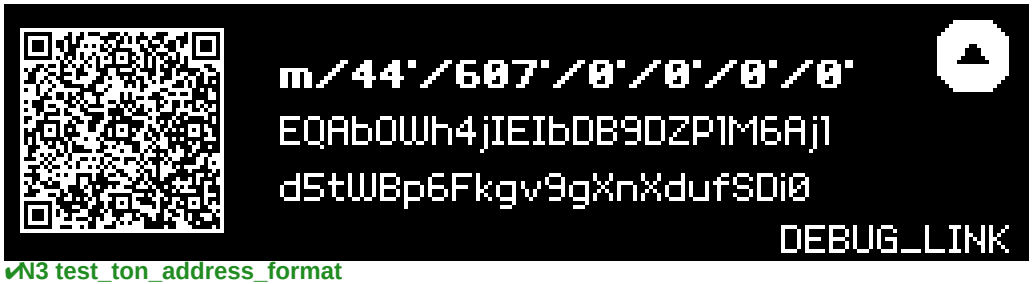
✔N2 test_ton_different_accounts

Different accounts (test_msg_ton_getaddress.py)
Different indices produce different addresses.

✔N2b test_ton_show_address

Show address on OLED (test_msg_ton_getaddress.py)
Full 48-char base64url TON address with QR code.



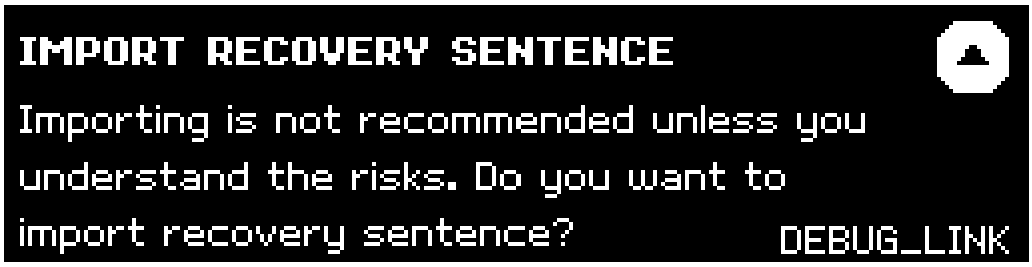


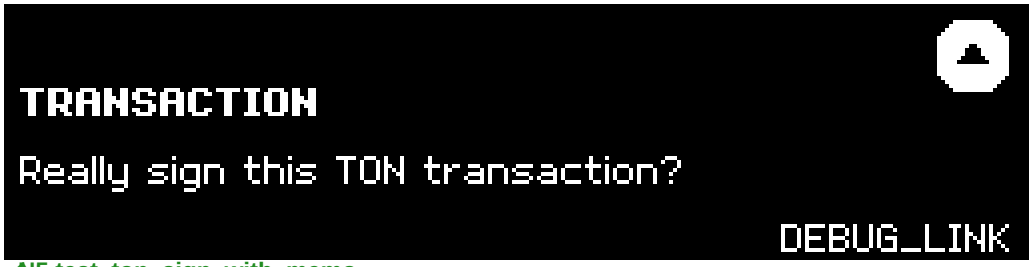
✔N3 test_ton_address_format

Address format validation (test_msg_ton_getaddress.py)
Bounceable/non-bounceable format check.

✔N4 test_ton_sign_structured

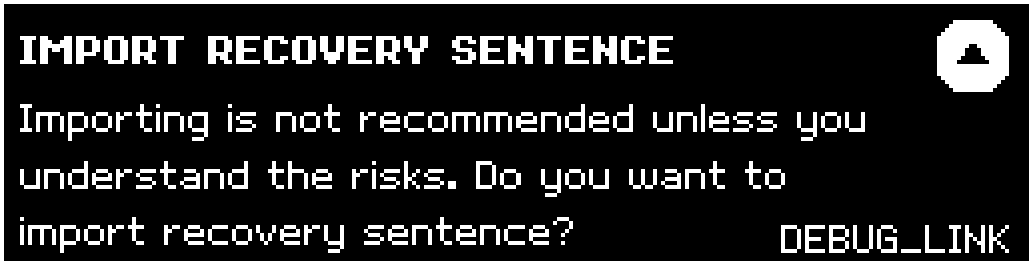
Sign TON transfer (test_msg_ton_signtx.py)
Structured fields shown as display context. Blind-sign with amount + address.

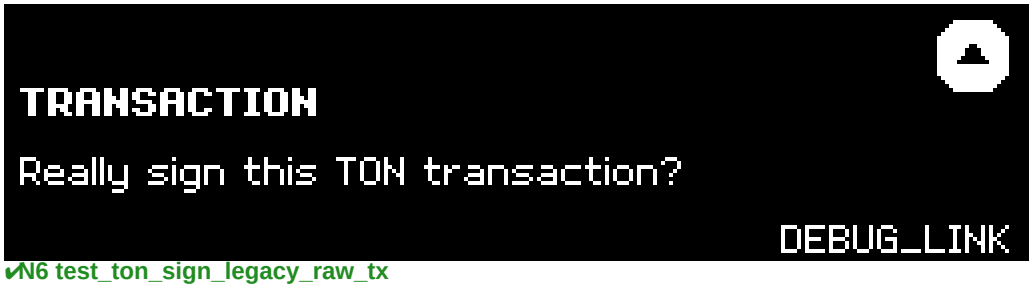




✔N5 test_ton_sign_with_memo

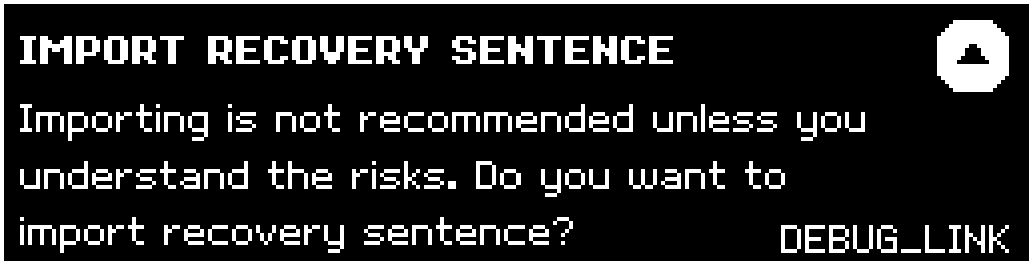
Sign TON with memo (test_msg_ton_signtx.py)
Memo/comment displayed before signing.

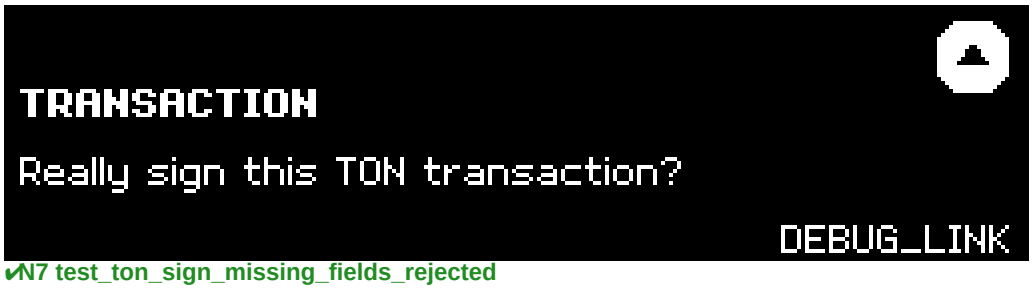




✔N6 test_ton_sign_legacy_raw_tx

Sign TON blind (test_msg_ton_signtx.py)
Raw tx without structured fields triggers blind sign.





✔N7 test_ton_sign_missing_fields_rejected

Missing fields rejected (test_msg_ton_signtx.py)
Incomplete data refused.

Z. Zcash Orchard [NEW]

NEW: Shielded transactions via PCZT streaming. Orchard hides sender, recipient, and amount using ZK proofs. Raw seed access (ZIP-32 Orchard derivation uses BIP-39 seed + Pallas curve). Full Viewing Key (FVK) export for watch-only wallets.

User Flow

FVK: Derive ak, nk, rivk components via ZIP-32 Orchard path
PCZT: Stream header -> actions one at a time -> confirm each -> return signatures
HYBRID: Transparent inputs + Orchard outputs in same tx

Tests: 9

-- Z1 test_fvk_reference_vectors

FVK reference vectors (test_msg_zcash_orchard.py)
FVK output matches known test vectors.
OLED needed: FVK export

-- Z2 test_fvk_field_ranges

FVK field ranges (test_msg_zcash_orchard.py)
ak, nk, rivk are within valid Pallas curve ranges.

-- Z3 test_fvk_consistency_across_calls

FVK deterministic (test_msg_zcash_orchard.py)
Same account always produces same FVK.

-- Z4 test_fvk_different_accounts

FVK different accounts (test_msg_zcash_orchard.py)
Different accounts produce different FVKs.

-- Z5 test_single_action_legacy_sighash

Sign single Orchard action (test_msg_zcash_sign_pczt.py)
One shielded action, device shows amount + fee.
OLED needed: Shielded confirm

-- Z6 test_multi_action_legacy_sighash

Sign multiple actions (test_msg_zcash_sign_pczt.py)
Multiple Orchard actions in one transaction.

-- Z7 test_signatures_are_64_bytes

Signature format (test_msg_zcash_sign_pczt.py)
Orchard signatures must be exactly 64 bytes (RedPallas).

-- Z8 test_transparent_shielding_single_input

Transparent to shielded (test_msg_zcash_sign_pczt.py)
Transparent BTC-like input shielded into Orchard pool.
OLED needed: Hybrid shield

-- Z9 test_transparent_shielding_multiple_inputs

Multi-input shielding (test_msg_zcash_sign_pczt.py)
Multiple transparent inputs shielded in one tx.

D. BIP-85 Child Derivation [NEW]

NEW: Derives child BIP-39 mnemonic from master seed via HMAC-SHA512 (BIP-85). Display-only: derived words appear on OLED, never transmitted over USB. Seed accessed in CONFIDENTIAL buffer, memzero'd after use.

User Flow

DERIVE: word_count + language + index -> HMAC-SHA512 -> child entropy -> BIP-39 words
DISPLAY: Words shown on OLED only -> user writes down -> never sent to host

Tests: 6

-- D1 test_bip85_12word_flow

Derive 12-word child (test_msg_bip85.py)
Derives 128 bits of child entropy -> 12-word BIP-39 mnemonic displayed on OLED.
OLED needed: Derivation params, Mnemonic on OLED

-- D2 test_bip85_24word_flow

Derive 24-word child (test_msg_bip85.py)
256 bits -> 24 words.

-- D3 test_bip85_18word_flow

Derive 18-word child (test_msg_bip85.py)
192 bits -> 18 words.

-- D4 test_bip85_different_indices_different_flows

Different indices (test_msg_bip85.py)
Index 0 and index 1 must produce completely different mnemonics.

-- D5 test_bip85_deterministic_flow

Deterministic (test_msg_bip85.py)
Same seed + same index always produces same child mnemonic.

-- D6 test_bip85_invalid_word_count

Invalid count rejected (test_msg_bip85.py)
Word counts other than 12/18/24 are refused.

Appendix: Device Specifications

The KeepKey is an open-source hardware wallet built on an ARM Cortex-M3 (STM32F205, 120MHz) with a 256x64 monochrome OLED, single confirmation button, and micro-USB interface. The bootloader (v2.x) is flashed at manufacture and never updated - it is the immutable root of trust. On every boot, the bootloader verifies the firmware signature using redundant F3 checks before transferring control.

BOOT SEQUENCE:
1. USB connect -> bootloader executes (always first)
2. F3 signature check (redundant dual-path verify)
3. Valid -> KeepKey logo -> firmware runs
4. Invalid/missing -> "UPDATE FIRMWARE" screen
5. Firmware upload -> verify -> flash -> reboot -> re-verify

HARDWARE:
- MCU: STM32F205RET6, 120MHz, 128KB bootloader + 896KB firmware
- Display: 256x64 OLED (SSD1306), monochrome, used for ALL confirmations
- Input: single capacitive button (confirm/reject)
- USB: micro-B, HID + WebUSB transports, HID fallback
- Storage: BIP-39 seed encrypted in isolated flash region

- Curves: secp256k1, ed25519, NIST P-256, Pallas (Zcash)

SECURITY MODEL:
- All private key operations happen on-device, keys never leave
- Every transaction output displayed on OLED for user verification
- PIN grid randomized on each prompt (position-based, not digit-based)
- BIP-39 passphrase creates hidden wallets (plausible deniability)