HENCEFORTH

A FORTH Programming Environment for iOS

+ Bitcoin SV Wallet

```
> : SQUARE DUP * ;
> 5 SQUARE .
25 ok.

> TX-NEW 5000 TX-SET-FEE
> TX-ADD-OUTPUT TX-BROADCAST
```

Henry Hudson

v1.0 · 2026

ok.

1

1.1 Links to Source Code

github.com/henryhudson/FORTHapp

Running the project requires Xcode on any iPhone / iPad device. Will soon be optimised for all Apple products.

Private for the time being – get in touch if you would like access.

1.2 Support

Follow us on Twitter

2

2.1 Achieved

- **Bitcoin (BSV) integration** — full wallet with HD key derivation (BIP-44 and Type42), transaction building, ARC broadcasting with failover, SPV verification, Paymail support, and 140+ Bitcoin Script opcodes including the Chronicle upgrade.
- **Block headers** — latest block height fetched on launch; SPV Merkle proof verification for transaction confirmation.
- **SwiftBSV** — fully integrated for transactions, Script building, BIP-32/39 key management, and ECDSA signing.
- **OPCodes** — all standard and Chronicle-restored opcodes available as FORTH words. Script Recording mode assembles Bitcoin Scripts from the terminal.
- **Files** — full CRUD for .fs and .forth files with iCloud backup. Script files identified by SCRIPT-BEGIN/SCRIPT-END markers.
- **Colour customisation** — foreground and background colour pickers, keyboard selection, and Dynamic Type support throughout.
- **Forth-2012 compliance** — 133/133 CORE words implemented, verified by the Hayes test suite.

2.2 Get People Coding

Provide a simple interface for the FORTH programming language, making it easy for people to learn to code.

2.3 Future

- Full sCrypt capability — each FORTH file will create a sCrypt that runs the opcodes within.
- CRDT (Conflict-free Replicated Data Type) system using the timestamp server for user-defined words.

- Trigonometry and calculus extensions.

3

3.1 Starting Forth – Leo Brodie

Leo Brodie, author of Starting Forth – a great textbook, suitable for both novices and professionals.

Starting FORTH (PDF)

3.2 BITSRFR

Fellow surfer of the binary ocean who has made FORTH tutorials and is making a Swift command line FORTH application.

YouTube · cruxFORTH on GitHub

3.3 Bitcoin

The greatest monetary system ever created. Invented by Craig Wright aka Satoshi Nakamoto – please read the white paper.

Bitcoin White Paper

4

Everything in FORTH is a **word** separated by spaces. That's all there is to the syntax. It's so simple.

> **FORTH is LIFO** – FORTH is a stack-based programming language which operates on a Last In, First Out principle.

4.1 Built-in Words

Ideally we give the user just enough to make use of FORTH and through FORTH some of Swift's functionality such as networking requests, the timer built into Swift, etc. But not so much as to be relied upon – user creation of words should be encouraged, and if possible a community database of words and their definitions be made.

4.1.1 Compile Words

One of the great things about FORTH is its ability to extend the compiler during run time.

Compile Operators

```
:           puts the user into compile mode where they can
            start to define their own word.
;           exits compile mode so that word interpretation
            can again occur.
recurse     calls the word being executed.
exit        exits the current word.
immediate   marks the most recently defined word as immediate,
            causing it to execute during compilation rather
            than being compiled.

postpone    forces compilation of the next word, even if it
            is marked IMMEDIATE. Used inside definitions to
            defer execution of immediate words.

value       (x --)
            Creates a named value that can be read by name
            and changed with TO.
            Usage: 42 value myval
```

```
to          (x --)
            Change the value of a word created with VALUE.
            Usage: 99 to myval

does>       ( -- )
            Defines the runtime behaviour of a CREATE'd word.
            Everything after DOES> runs when the created word
            is executed, with the data-field address on the
            stack.
            Usage: : array create cells allot does> swap
            cells + ;

include     ( -- )
            Loads and runs a FORTH file.
            Usage: include maths.fs
```

4.1.2 Meta Programming

Meta Programming

```
'           ( -- xt)
            Gets the execution token of the named word. Used to
            get a reference to a word for later execution.

[']         ( -- xt)
            Compile-time version of ' (tick). Compiles the
            execution token of a word into the current definition.

execute     (xt --)
            Executes the word whose execution token is on the
            stack. Enables dynamic word execution.

literal     (x --)
            Compiles a numeric literal into the current word
            definition during compilation.

state       ( -- addr)
            Returns the address of the STATE variable. STATE is 0
            when interpreting, -1 when compiling.

[           (--)
            Switches from compilation mode to interpretation mode.

]           (--)
            Switches from interpretation mode to compilation mode.

evaluate    (c-addr u --)
            Interprets the string at c-addr of length u as if
            it were typed at the terminal.
```

```
find      (c-addr -- c-addr 0 | xt 1 | xt -1)
          Searches the dictionary for the counted string at
          c-addr. Returns xt and 1 (immediate) or -1 (normal),
          or c-addr and 0 if not found.

>body     (xt -- a-addr)
          Returns the data-field address of a CREATE'd word
          given its execution token.

>in       ( -- a-addr)
          Returns the address of the variable holding the
          current input parse position.

>number   (ud1 c-addr1 u1 -- ud2 c-addr2 u2)
          Converts characters at c-addr to a number using
          the current BASE. Stops at the first non-convertible
          character.
```

4.1.3 Arithmetic Operators

Arithmetic is done using reverse Polish notation, so the numbers are required on the stack before the arithmetic operator can be used.

Arithmetic Operators

```
+         (n1 n2 -- sum)
          Takes the top two items from the stack and returns
          their sum to the stack.

-         (n1 n2 -- difference)
          Takes the top two items from the stack and returns
          their difference to the stack.

*         (n1 n2 -- product)
          Takes the top two items from the stack and returns
          their product to the stack.

/         (n1 n2 -- quotient)
          Takes the top two items from the stack and returns
          their quotient to the stack.

*/        (n1 n2 n3 -- result)   result = (n1*n2/n3)
          Takes the top three items from the stack. The first
          two are multiplied together, then divided by the last
          item. The result is appended to the stack.

mod       (n1 n2 -- modulus)
          Takes two items from the stack, divides the first by
          the second and the remainder is added to the stack.

/mod      (n1 n2 -- modulus quotient)
          Takes the last two items from the stack and returns
```

```
          their modulus and their quotient to the stack.

abs       (n -- |n|)
          Takes the last value from the stack and returns its
          absolute value.

negate    (n -- -n)
          Takes the last item from the stack and returns its
          value negated.

max       (n1 n2 -- max)
          Takes the last two values from the stack and returns
          the greatest value to the stack.

min       (n1 n2 -- min)
          Takes the last two values from the stack and returns
          the smallest value to the stack.

1+        (n -- n+1)
          Takes the top item from the stack, adds one to it and
          returns the result to the stack.

1-        (n -- n-1)
          Takes the top item from the stack, minuses one from it
          and returns the result to the stack.

2+        (n -- n+2)
          Takes the top item from the stack, adds two to it and
          returns the result to the stack.

2-        (n -- n-2)
          Takes the top item from the stack, minuses two from it
          and returns the result to the stack.

2*        (n -- n*2)
          Takes the top item from the stack, multiplies it by
          two and returns the result to the stack.

2/        (n -- n/2)
          Takes the top item from the stack, halves it and
          returns the result to the stack.

sumStack (n1 ... n -- sum)
          Adds all items on the stack and appends the sum.

*/mod     (n1 n2 n3 -- remainder quotient)
          Multiplies n1 by n2, then divides by n3. Returns
          both the remainder and the quotient.
```

4.1.4 Double-Cell Arithmetic

Double-cell words operate on 64-bit values represented as two stack items (high cell on top). These are required by the Forth-2012 CORE standard for intermediate precision

in calculations.

Double-Cell Arithmetic

```
s>d       (n -- d)
          Sign-extends a single-cell number to a double-cell
          number. Required before FM/MOD or SM/REM.

m*        (n1 n2 -- d)
          Signed multiply of n1 by n2, producing a double-cell
          result. No overflow possible.

um*       (u1 u2 -- ud)
          Unsigned multiply of u1 by u2, producing an unsigned
          double-cell result.

fm/mod    (d n1 -- n2 n3)
          Floored division of double-cell d by n1. The quotient
          n3 rounds toward negative infinity. The remainder n2
          has the same sign as the divisor.

sm/rem    (d n1 -- n2 n3)
          Symmetric division of double-cell d by n1. The
          quotient n3 rounds toward zero. The remainder n2 has
          the same sign as the dividend.

um/mod    (ud u1 -- u2 u3)
          Unsigned division of unsigned double-cell ud by u1.
          Returns remainder u2 and quotient u3.
```

4.1.5 Numeric Output

Numeric Output

```
<#        (--)
          Initialises pictured numeric output conversion.

#         (ud1 -- ud2)
          Converts one digit from the double number and adds it
          to the pictured numeric output buffer.

#s        (ud -- 0 0)
          Converts all remaining digits from the double number
          until it becomes zero.

#>        (xd -- c-addr u)
          Completes pictured numeric conversion and returns the
          address and length of the formatted string.

hold      (char --)
          Adds a character to the beginning of the pictured
          numeric output buffer.
```

```
sign     (n --)
         If the number is negative, adds a minus sign to the
         pictured numeric output.

.r       (n1 n2 --)
         Displays a number right-aligned in a field of n2
         characters width.

u.       (u --)
         Displays an unsigned number followed by a space.

u.r      (u n --)
         Displays an unsigned number right-aligned in a field
         of n characters width.
```

4.1.6 Comparison Operators

> f is a flag or bool [true | false]. Per the Forth-2012 standard, TRUE is -1 (all bits set) and FALSE is 0. Any non-zero value is considered true by IF, AND, OR, and other conditional words.

Comparison Operators

```
=     (n1 n2 -- f)
      Takes the top two values from the stack, checks if they
      are equal, then returns the flag to the stack.

<>    (n1 n2 -- f)
      Takes the top two values from the stack, checks if they
      are NOT equal, then returns the flag to the stack.

<     (n1 n2 -- f)
      Takes the top two values from the stack, checks if n1 is
      less than n2, then returns the flag to the stack.

>     (n1 n2 -- f)
      Takes the top two values from the stack, checks if n1 is
      greater than n2, then returns the flag to the stack.

&&    (n1 n2 -- f)
      Logical AND. Returns TRUE (-1) if both values are
      non-zero, FALSE (0) otherwise.

||    (n1 n2 -- f)
      Logical OR. Returns TRUE (-1) if either value is
      non-zero, FALSE (0) otherwise.

0=    (n1 -- f)
      Takes the top value from the stack, checks if it is equal
```

```
        to zero, then returns the flag to the stack.

0<      (n1 -- f)
        Takes the top value from the stack, checks if it is less
        than zero, then returns the flag to the stack.

0>      (n1 -- f)
        Takes the top value from the stack, checks if it is
        greater than zero, then returns the flag to the stack.

u<      (u1 u2 -- f)
        Unsigned comparison. Returns true if u1 is less than u2
        when both are treated as unsigned values.
```

4.1.7 Bitwise Operators

Bitwise Operators

```
and     (n1 n2 -- AND)
        Takes the top two values from the stack and returns the
        result of their logical AND to the stack.

or      (n1 n2 -- OR)
        Takes the top two values from the stack and returns the
        result of their logical OR to the stack.

xor     (n1 n2 -- XOR)
        Takes the top two values from the stack and returns the
        result of their logical XOR to the stack.

invert  (n -- ~n)
        Bitwise inversion. Flips every bit of the top value
        on the stack.

rshift  (n1 n2 -- n3)
        Takes the top two values from the stack and returns the
        result of n1 bitwise right-shifted by n2.

lshift  (n1 n2 -- n3)
        Takes the top two values from the stack and returns the
        result of n1 bitwise left-shifted by n2.
```

4.1.8 Stack Operators

Stack Operators

```
.s              ( -- )
                Displays the current stack to the terminal.

.               (n -- )
                Takes the last value from the stack and pops it
                to terminal.

dup             (n -- n n)
                The top value is taken from the stack, duplicated
                and both the original and copy returned.

?dup            (n -- 0 | n n)
                Duplicates the top item on the stack only if it
                is non-zero.

nip             (n1 n2 -- n2)
                The first item below the top of the stack is
                dropped.

swap            (n1 n2 -- n2 n1)
                The last two values are taken from the stack,
                swapped and then returned to the stack.

drop            (n -- )
                Discards the top item from the stack.

2dup            (n1 n2 -- n1 n2 n1 n2)
                The top two values from the stack are taken,
                copied and both original and copy returned.

over            (n1 n2 -- n1 n2 n1)
                The second value from the stack is copied and
                added to the stack.

pick            (nx ... n x -- nx ... n nx)
                Takes the top value from the stack and copies the
                nth value, appending it to the end of the stack.

rot             (n1 n2 n3 -- n2 n3 n1)
                The top three values of the stack are taken and
                rotated left then returned to the stack.

-rot            (n1 n2 n3 -- n3 n1 n2)
                The top three values of the stack are taken and
                rotated right then returned to the stack.

tuck            (n1 n2 -- n2 n1 n2)
                The top two values from the stack are taken. The
                top value is copied and placed before the second
                value in the stack (third value deep).
```

```
2drop          (n1 n2 -- )
               The top two values are taken from the stack and
               discarded.

2swap          (n1 n2 n3 n4 -- n3 n4 n1 n2)
               The top four values are taken from the stack, and
               the order of the two pairs is reversed.

2over          (n1 n2 n3 n4 -- n1 n2 n3 n4 n1 n2)
               The first four values are taken from the stack,
               with items 3 and 4 deep being copied and added.

reverseStack   (n1 n2 ... nlast -- nlast ... n2 n1)
               The first shall become last and the last shall
               become first. The stack's order is reversed.

depth          ( -- n)
               Returns the number of items currently on the data
               stack before depth was executed.

roll           (xu xu-1 ... x0 u -- xu-1 ... x0 xu)
               Rotates u+1 items on the stack. The u-th item is
               moved to the top of the stack.

stackCount     ( -- n)
               The number of items on the stack is added to the
               stack.
```

4.1.9 Return Stack Operators

Return Stack Operators

```
>r    (n -- )
      Takes top value from the parameter stack and moves it to
      the return stack.

r>    ( -- n)
      Takes top value from the return stack and moves it to the
      parameter stack.

r@    ( -- n)
      Copies last item from return stack.

i     ( -- n)
      Copies top item in return stack.

i'    ( -- n)
      Copies second item in return stack.

j     ( -- n)
```

```
      Copies third item in return stack.

nth   ( -- n)
      Copies the nth item in the return stack.
```

4.1.10 Decision Operators

> f is a flag or bool [true | false].

Decision Operators

```
if      (f -- )
        Takes a flag from the stack and executes words if true
        flag.

else    (f -- )
        If flag taken from if was false these words will be
        executed. This is optional.

then    (f -- )
        Denotes the end of the if statement.

not     (f -- f)
        Takes a flag value and reverses its result.

within (n1 n2 n3 -- f)
        Takes three values from the stack and checks if n2 is a
        number within n1 and n3.

true    ( -- -1)
        Pushes the true flag (-1, all bits set) to the stack.
        Per Forth-2012 standard.

false   ( -- 0)
        Pushes the false flag (0) to the stack.

case    ( -- )
        Begins a CASE...OF...ENDOF...ENDCASE switch statement.
        The selector value must be on the stack.

of      (x1 x2 -- | x1)
        Compares the top of stack with the selector. If equal,
        drops both and executes the clause. If not, skips to
        ENDOF.

endof   ( -- )
        Ends an OF clause. Branches to after ENDCASE.

endcase (x -- )
        Ends a CASE statement. Drops the selector value if no
```

```
        OF clause matched.

catch   (xt -- exception# | 0)
        Executes the word referred to by xt. If THROW is called
        during execution, catches the exception and returns
        the error code. Returns 0 on success.

throw   (0 | exception# -- )
        Throws an exception. If 0, does nothing. A non-zero
        value unwinds the stack to the nearest CATCH.
```

4.1.11 Loop Operators

Loop Operators

```
do      (n1 n2 --)
        n1 is endValue, n2 is startValue. Repeats words between
        do and loop from startValue until endValue is met.

loop    ( -- )
        Ends loop.

+loop   (n1 -- )
        Loops in n1 increments.

every   (n1 -- )
        Performs word after every, every n1 seconds.

stop    ( -- )
        Stops every running.

begin   ( f -- )
        Marks the start of a while loop.

while   ( f -- )
        Tests a condition within a BEGIN...REPEAT loop. If the
        condition is false, exits the loop.

repeat  ( -- )
        Marks the end of a BEGIN...WHILE...REPEAT loop and jumps
        back to BEGIN.

until   ( -- )
        Ends BEGIN loop when condition is met.

leave   ( -- )
        Exits the current DO...LOOP or DO...+LOOP immediately.
        Must be used within a loop.

unloop  ( -- )
        Discards the loop control parameters from the return
```

```
        stack. Used when exiting a loop early with LEAVE.

?do     (n1 n2 --)
        Like DO but skips the loop entirely if the start and
        end values are equal. Executes loop from n1 to n2-1.

again   ( -- )
        Unconditional branch back to BEGIN. Creates an infinite
        loop (BEGIN...AGAIN). Use LEAVE or the Escape key to
        exit.
```

4.1.12 Base Operations

Much like FORTH, Henceforth accepts integers as binary, octal, decimal and hexadecimal. Just be sure to change the base.

Base Operations

```
base      ( -- addr)
          Returns the address of the BASE variable (Forth-2012
          compliant). Use BASE @ to read and n BASE ! to write.
          Supports bases 2 through 36.

decimal   (--)
          Sets BASE to 10.

binary    (--)
          Sets BASE to 2.

octal     (--)
          Sets BASE to 8.

hex       (--)
          Sets BASE to 16.
```

4.1.13 Pop-up Operations

Pop-up Operations

```
lineChart     (--)
              Displays a line chart representing the stack.

barChart      (--)
              Displays a bar chart representing the stack.

pieChart      (--)
              Displays a pie chart representing the stack with
              slices in proportion to its value's weight compared
              to the stack's sum.
```

```
applemap     ( -- )
             Presents Apple Maps view.
```

4.1.14 Constants and Variables

Constants and Variables

```
constant (n --)
         Takes a value from the stack and a name and creates a
         constant of that value called by the name provided.

variable (--)
         Creates a variable whose name will be the first string
         of characters following variable, initially zero.

!        (n address --)
         Stores a number to the address provided.

@        (address -- n)
         Gives the value from address.

?        (address --)
         Prints the value from address to the terminal.

c!       (char addr --)
         Stores a single character at the specified memory
         address.

c@       (addr -- char)
         Fetches a single character from the specified memory
         address and places it on the stack.

create   (--)
         Creates an array named by the first string of
         characters following the create word.

,        (n --)
         Takes the top two values from the stack and stores the
         second value as part of the array in the address of
         the first value.

allot    (n --)
         Allocates n cells of data space. Used after CREATE to
         allocate memory for arrays or data structures.

here     ( -- addr)
         Returns the address of the next available data space
         location.

cells    (n1 -- n2)
```

```
            Converts a count to an address offset by multiplying
            by the cell size.

cell+       (addr1 -- addr2)
            Adds the size of one cell to an address, advancing it
            to the next cell boundary.

+!          (n addr --)
            Adds n to the value stored at the given address.

count       (c-addr1 -- c-addr2 u)
            Converts a counted string (length byte followed by
            characters) to an address and length suitable for TYPE.

bl          ( -- char)
            Pushes the ASCII value for space character (32) onto
            the stack.

2@          (addr -- x1 x2)
            Fetches two consecutive cells from memory at the given
            address.

2!          (x1 x2 addr --)
            Stores two consecutive values to memory at the given
            address.

fill        (addr u char --)
            Fills u bytes of memory starting at addr with the
            specified character value.

move        (addr1 addr2 u --)
            Copies u address units from addr1 to addr2. Handles
            overlapping memory regions correctly.

erase       (addr u --)
            Fills u address units with zeros, effectively clearing
            the memory region.

allocate    (u -- a-addr ior)
            Allocates u cells of contiguous data space. Returns the
            start address and an I/O result code (0 = success, -1 =
            failure). Reuses previously freed blocks when possible.

free        (a-addr -- ior)
            Releases previously allocated memory starting at addr.
            Returns 0 on success. Adjacent freed blocks are merged
            to reduce fragmentation.

resize      (a-addr u -- a-addr2 ior)
            Changes the size of an allocation. If the new size fits
            within the existing block, returns the same address.
            Otherwise allocates a new block, copies existing data,
            and frees the old block.
```

```
c,        (char --)
          Reserves one character of data space and stores char
          in it.

char+     (c-addr1 -- c-addr2)
          Adds the size of one character to the address.

chars     (n1 -- n2)
          Returns the size in address units of n characters.
          In this system, 1 char = 1 unit.

align     ( -- )
          Aligns the data-space pointer to a cell boundary.

aligned   (addr -- a-addr)
          Returns the first aligned address greater than or
          equal to addr.
```

4.1.15 Terminal Operators

Terminal Operators

```
cr            (--)
              Newline added to terminal.

space         (--)
              Adds a single space to terminal.

spaces        (n --)
              Takes a value from the stack and adds that number
              of spaces to the terminal.

emit          (n --)
              Takes a value from the stack and emits its ASCII
              code.

(             (--)
              Starts inline comment. Everything between ( and )
              will be ignored. Can appear anywhere on a line.

)             (--)
              Ends inline comment.

\textbackslash            (--)
              Line comment. Everything after \textbackslash{} to
              the end of the line is ignored. Supported in .fs
              files. Example: \textbackslash{} this is a comment

.s            (--)
              Show stack on terminal.
```

```
.rs             (--)
                Shows return stack.

clear           (--)
                Clears stack and return stack.

bye             (--)
                Clears everything, resets to default settings.

."              (--)
                Start print. Everything between quotes is printed.

"               (--)
                Ends print.

s"              ( -- addr u)
                Creates a string literal. Returns the address and
                length of the string on the stack.

type            (addr u --)
                Takes an address and length from the stack and
                displays the string at that address to the terminal.

page            (--)
                Adds 24 new lines to the terminal.

char            ( -- char)
                Parses the next space-delimited word and returns
                the ASCII value of its first character.

[char]          ( -- n)
                Compile-time version of CHAR. Compiles the
                character value into the current definition.

word            (char -- c-addr)
                Parses the input stream until the delimiter
                character is found and returns a counted string
                address.

parse           (char -- c-addr u)
                Parses the input stream until the delimiter
                character is found and returns address and length.

key             ( -- n)
                Waits for a single keypress and pushes its ASCII
                value onto the stack. Blocks execution until a key
                is pressed. Works with both hardware and software
                keyboards.

?terminal       ( -- flag)
                Returns true if the keyboard buffer has input or
                if a terminal interrupt has been requested (break
                key pressed), false otherwise. FIG-FORTH semantics.
```

```
accept        (c-addr n1 -- n2)
              Receives up to n1 characters from the input source
              and stores them at the buffer address c-addr.
              Returns n2, the actual number of characters
              received.

sessionInput  (--)
              Displays session input.

dictionary    (--)
              Shows the user's defined words, constants and
              variables.

vlist         (--)
              Displays a list of all the built-in words
              Henceforth offers.

forth         (--)
              Resets vocabulary to standard built-in words,
              removing all user definitions.

c"            ( -- c-addr)
              Creates a counted string literal. Returns the
              address of a counted string (length byte followed
              by characters).

abort         (--)
              Clears all stacks and resets the interpreter state.

abort"        (flag --)
              If the flag is non-zero, displays the message and
              ABORTs. Otherwise discards the message and
              continues.
              Usage: 0= abort" value must not be zero"

quit          (--)
              Clears the return stack and enters interpretation
              state.

source        ( -- c-addr u)
              Returns the address and length of the current
              input buffer.

environment?  (c-addr u -- false | i*x true)
              Queries the system for an environmental parameter.
              Returns false if unknown, or the value and true
              if known.
```

4.1.16 Other Words

Other Words

```
date&time (-- second minute hour day month year)
          Adds the device's current time to the stack.

pay       (--)
          Presents pay sheet, where you can send bitcoin.

.fs       (--)
          Filename followed by .fs extension will load that
          file's contents to Henceforth.

startup.fs
          A special file that runs automatically when
          Henceforth launches. Created as a blank template
          on first launch. Add your custom word definitions,
          aliases, and INCLUDE calls here so they are
          available every session.
```

4.1.17 Data Conversion

Henceforth has two stacks: the integer stack (main) and the data stack (for raw bytes, used by Bitcoin Script operations). These words bridge between them.

Data Conversion

```
hex2data    ( -- )
            Converts the following hex string to bytes and
            pushes them onto the data stack.
            Usage: hex2data 48656c6c6f

data2hex    ( -- )
            Converts the top data stack item to a hex string
            and displays it on the terminal.

.ds         ( -- )
            Displays the contents of the data stack.

>data       ( -- )
            Converts the following token to raw data bytes and
            pushes them onto the data stack.

>addr       (n -- )
            Marks the top-of-stack integer value as a Bitcoin
            address string for transaction building.
```

4.1.18 String Operations

String Operations

```
cmove       (c-addr1 c-addr2 u -- )
            Copies u characters from addr1 to addr2, proceeding
            low address to high (safe when addr2 > addr1).

cmove>      (c-addr1 c-addr2 u -- )
            Copies u characters from addr1 to addr2, proceeding
            high address to low (safe for overlapping regions
            when addr2 < addr1).

/string     (c-addr1 u1 n -- c-addr2 u2)
            Adjusts a string address and length by n characters.
            Advances the address by n and reduces the length by
            n.

search      (c-addr1 u1 c-addr2 u2 -- c-addr3 u3 flag)
            Searches for string2 within string1. If found,
            returns the match address, remaining length, and
            true. If not found, returns the original string and
            false.

blank       (c-addr u -- )
            Fills u characters of memory starting at addr with
            spaces (BL, ASCII 32).
```

4.1.19 Networking Operations

Networking Operations

```
wocbitcoinprice         ( -- n)
                        Fetches the bitcoin price in USD and
                        adds it to the stack in pennies.

woccirculatingsupply    (--)
                        Fetches the circulating supply of
                        bitcoin.

wocblockchaininfo       (--)
                        Fetches latest general block info from
                        WhatsOnChain and prints to terminal.

wocaddressinfo          (--)
                        Takes an address and displays its
                        information to the terminal.

wocgetbalance           (--)
                        Takes an address and displays its
                        balance information to the terminal.
```

```
wocgethistory           (--)
                        Takes an address and displays its
                        history information to the terminal.

wocgetblockbyheight     (n--)
                        Takes a block height from the stack and
                        searches WhatsOnChain for info on that
                        block.
```

4.1.20 Transaction Builder

The transaction builder words allow programmatic construction of Bitcoin transactions directly from the FORTH terminal. Build a transaction step by step, then broadcast it.

Transaction Builder

```
tx-new          ( -- )
                Initialises a new transaction builder. Clears
                any previous transaction state.

tx-set-fee      (n -- )
                Sets the fee rate in satoshis per kilobyte for
                the current transaction.
                Usage: 500 tx-set-fee

tx-add-output   (n -- )
                Adds a P2PKH output to the transaction. Expects
                the amount in satoshis on the stack and the
                destination address as the next word.
                Usage: 1000 tx-add-output 1A1zP1...

tx-add-data     ( -- )
                Adds an OP\_RETURN data output to the current
                transaction.

tx-preview-on   ( -- )
                Enables preview mode --- TX-BROADCAST will show
                the transaction without actually broadcasting.

tx-preview-off  ( -- )
                Disables preview mode --- TX-BROADCAST will
                broadcast to the network.

tx-show         ( -- )
                Displays the current transaction being built
                (inputs, outputs, fee).

tx-clear        ( -- )
                Clears the transaction builder, discarding all
                inputs and outputs.
```

```
tx-broadcast    ( -- )
                Builds, signs, and broadcasts the current
                transaction to the BSV network via ARC.

send            (n -- )
                Standard payment in a single word. Reads the
                destination address(es) from the last ." string
                and the amount in satoshis from the stack.
                Auto-selects UTXOs from the default wallet,
                uses the wallet's fee rate, and broadcasts
                via ARC.
                Usage: ." 1A1zP1..." 1000 send
```

Transaction Builder Example: Send 1000 satoshis to an address with a custom fee rate using the step-by-step builder:

```
tx-new
250 tx-set-fee
." 1ExampleAddress..." 1000 tx-add-output
tx-show
tx-broadcast
```

Standard Payment Example: The send word wraps the entire PayView payment flow into a single command. It uses the default wallet (marked with a star on the wallet card):

```
\ Single recipient:
." 1A1zP1eP5QGefi2DMPTfTL5SLmv7DivfNa" 1000 send

\ Multiple recipients (space-separated):
." 1A1zP1... 1BvBMS..." 5000 send
```

Addresses are wrapped in ." (dot-quote), which is FORTH's string literal syntax. The address string is placed on the terminal output, and send reads it from there. Multiple addresses can be space-separated within a single ." string for multi-output transactions.

Standard Transaction Script

Every installation includes a standard-transaction.fs file in the Files tab. This is a Bitcoin Script file that defines the OP_RETURN output attached to your payments. It is automatically loaded each time you open the Pay view.

```
( Standard Transaction Script )

SCRIPT-BEGIN
OP_FALSE OP_RETURN
." Hello, you are welcome." >data data>script
SCRIPT-END
```

How it works:

1. SCRIPT-BEGIN activates script recording mode
2. OP_FALSE OP_RETURN emits the opcodes for a data-carrying output
3. ." places the message on the terminal output

4. >data converts it to raw bytes on the data stack
5. data>script emits those bytes as PUSHDATA into the script buffer
6. SCRIPT-END finalises the compiled script

To customise your standard transaction, edit the message string in the file. The Pay view re-reads the file from disk each time it opens, so edits take effect immediately. The file also appears in the Script tab of the script picker if you need to switch between scripts.

> **Note on comments:** Script files use ( ) parenthetical comments rather than \ line comments. The script picker flattens newlines when compiling, which causes \ comments to consume the rest of the file.

4.1.21 Bitcoin Cryptography

Bitcoin Cryptography

```
sign-msg      ( -- )
              Signs a message with the current wallet key using
              Bitcoin Signed Message (BSM).
              Usage: sign-msg <message>

verify-msg    ( -- flag)
              Verifies a BSM signature against an address. Pushes
              true (-1) or false (0) to stack.
              Usage: verify-msg <address> <signature> <message>

encrypt-msg   ( -- )
              Encrypts a message for a recipient using BRC-2
              Electrum ECIES (ECDH + AES-CBC + HMAC-SHA256).
              Wire-compatible with @bsv/sdk EncryptedMessage.
              Usage: encrypt-msg <recipientPubKeyHex> <message>

decrypt-msg   ( -- )
              Decrypts a BRC-2 ECIES-encrypted message using the
              current wallet key. The plaintext is displayed.
              Usage: decrypt-msg <ciphertextHex>
```

4.1.22 BAP Identity

BAP (Bitcoin Attestation Protocol) provides a decentralised identity system built on Bitcoin. Identity keys are derived using Type42 (BRC-42) key derivation from the wallet's master key.

BAP Identity Operations

```
bap-id        ( -- )
              Shows the BAP identity key (ID) and signing address
```

```
                for the current wallet.

bap-sign        ( -- )
                Signs a message with the BAP identity key (Type42
                derived). Usage: bap-sign <message>

bap-verify      ( -- flag)
                Verifies a BAP-signed message. Pushes true (-1) or
                false (0) to the stack.
                Usage: bap-verify <address> <signature> <message>

bap-register    ( -- )
                Builds a BAP ID registration transaction
                (OP_RETURN with BAP + AIP attestation).
```

4.1.23 Script Recording

Script recording is a macro assembler for Bitcoin Script. When recording is active, op_* words emit their opcodes into a script buffer rather than executing. Standard FORTH control flow still runs normally, allowing programmatic script construction.

Script Recording Operations

```
script-begin    ( -- )
                Start recording Bitcoin Script. op_* words emit
                opcodes to the script buffer. FORTH control flow
                still runs normally.

script-end      ( -- )
                Stop recording and finalise the script. Displays
                the compiled script hex.

script-show     ( -- )
                Display the current script buffer contents as hex.

script-clear    ( -- )
                Discard the current script buffer and stop
                recording.

script-use      ( -- )
                Set the recorded script as the selected transaction
                script (for building transactions).

script-save     ( -- )
                Save the current script with a name.
                Usage: script-save <name>

script-load     ( -- )
                Load a previously saved script by name.
                Usage: script-load <name>
```

```
script-list    ( -- )
               List all saved scripts.
```

4.1.24 Script Bridge Words

The bridge words connect FORTH's stacks to the script recording buffer, enabling dynamic script construction with computed values, strings, and file data.

Script Bridge Operations

```
>script        (n -- )
               Takes an integer from the stack and emits it as a
               Bitcoin Script number to the script buffer.

data>script    ( -- )
               Takes the top item from the data stack and emits
               it as PUSHDATA to the script buffer. Use after
               crypto operations to inject computed hashes.

text>script    ( -- )
               Converts the following token to UTF-8 bytes and
               emits as PUSHDATA to the script buffer.
               Usage: text>script image/jpeg

file>script    ( -- )
               Reads a file's raw contents and emits them as
               PUSHDATA to the script buffer. For uploading
               images, videos, or data on-chain.
               Usage: file>script photo.jpg
```

4.1.25 Script Helper Words

Pre-built script templates for common Bitcoin transaction patterns.

Script Helpers

```
make-p2pkh-script      ( -- )
                       Creates a standard P2PKH (Pay-to-Public-
                       Key-Hash) locking script from an address.

make-opreturn-script   ( -- )
                       Creates an OP\_RETURN output script from
                       data on the stack.

make-2of3-multisig     ( -- )
                       Creates a 2-of-3 multisig script from
                       three public keys on the stack.

script-pushdata        ( -- )
```

```
Pushes raw data bytes to the script
recording buffer using OP\_PUSHDATA.
```

Example: Upload an image to the blockchain using the B:// protocol:

```
script-begin
  op_false op_return
  text>script 19HxigV4QyBv3tHpQVcUEQyq1pzZVdoAut
  file>script photo.jpg
  text>script image/jpeg
  text>script binary
  text>script my_photo
script-end
script-use
pay
```

Example: Build a hash-locked script with a computed HASH160:

```
script-begin
  op_dup op_hash160
  " secret" op_sha256 op_hash160
  data>script
  op_equalverify op_checksig
script-end
```

Here data>script bridges the data stack (where the hash was computed) into the script buffer — something impossible without the bridge words.

Example: Use a FORTH loop to generate repetitive script patterns:

```
script-begin
  10 0 do  op_1 op_add  loop
script-end
```

This emits 10 copies of OP_1 OP_ADD into the script. FORTH's full control flow (loops, variables, conditions) runs normally during recording — only op_* words emit to the buffer.

Example: Build a simple P2PKH locking script:

```
script-begin op_dup op_hash160 <pubKeyHash> op_equalverify op_checksig
    script-end
```

Scripts saved with script-save appear in the Files tab with an orange icon and can be selected as transaction scripts via script-use or the Script pop-up.

4.1.26 OPCodes

Push Data OP Codes

```
op_0            ( -- 0)       Zero is pushed on to the stack.
op_1            ( -- 1)       One is pushed on to the stack.
op_2            ( -- 2)       Two is pushed on to the stack.
op_3            ( -- 3)       Three is pushed on to the stack.
op_4            ( -- 4)       Four is pushed on to the stack.
op_5            ( -- 5)       Five is pushed on to the stack.
op_6            ( -- 6)       Six is pushed on to the stack.
op_7            ( -- 7)       Seven is pushed on to the stack.
op_8            ( -- 8)       Eight is pushed on to the stack.
op_9            ( -- 9)       Nine is pushed on to the stack.
op_10           ( -- 10)      Ten is pushed on to the stack.
op_11           ( -- 11)      Eleven is pushed on to the stack.
op_12           ( -- 12)      Twelve is pushed on to the stack.
op_13           ( -- 13)      Thirteen is pushed on to the stack.
op_14           ( -- 14)      Fourteen is pushed on to the stack.
op_15           ( -- 15)      Fifteen is pushed on to the stack.
op_16           ( -- 16)      Sixteen is pushed on to the stack.

op_false        ( -- 0)       Zero (false flag) pushed to stack.

op_pushdata1    (--)    The byte after will contain how many
                        bytes are to be pushed on to the stack.

op_pushdata2    (--)    The next two bytes will contain how many
                        bytes will be pushed on to the stack.

op_pushdata4    (--)    The next four bytes will contain how many
                        bytes will be pushed on to the stack.

op_1negate      ( -- -1)    Minus one pushed on to the stack.
```

Control Flow OP Codes

```
op_nop          (--)
                Does nothing.

op_ver          ( -- version)
                Pushes the transaction version onto the stack.
                Returns 1 (legacy) or 2 (Chronicle, post block
                943,816).

op_if           (f--)
                Executes code depending on the expression's flag.

op_notif        (n--)
                Executes code depending on the expression's flag.

op_verif        ( threshold -- flag)
                Pushes TRUE if tx version >= threshold, FALSE
```

```
                otherwise. Chronicle upgrade.

op_vernotif     ( threshold -- flag)
                Pushes TRUE if tx version < threshold, FALSE
                otherwise. Chronicle upgrade.

op_else         (--)
                Executes if the preceding OP_IF, OP_NOTIF, or
                OP_ELSE were not executed.

op_endif        (--)
                Ends the if/else statement.

op_verify       (f--)
                Marks the transaction as INVALID if the flag is
                false.

op_return       (--)
                Ends script. Often used to add data to the
                blockchain.
```

Stack Operator OP Codes

```
op_toaltstack       (n--)
                    Takes the top value from the stack and pushes
                    it to the alt stack.

op_fromaltstack     (--n)
                    Takes the top value from the alt stack and
                    pushes it to the stack.

op_2drop            (n n1--)
                    Discards the top 2 elements from the stack.

op_2dup             (n n1 -- n n1 n n1)
                    Duplicates the top two values from the stack
                    adding back both original and copy.

op_3dup             (n n1 n2 -- n n1 n2 n n1 n2)
                    Duplicates the top three values from the stack
                    adding back both original and copy.

op_2over            (n n1 n2 n3 -- n n1 n2 n3 n n1)
                    Copies the fourth and third items in the stack
                    over the first and second.

op_2rot             (n n1 n2 n3 n4 n5 -- n2 n3 n4 n5 n n1)
                    Copies the fifth and sixth items in the stack
                    over the first and second.

op_2swap            (n n1 n2 n3 -- n2 n3 n n1)
                    Takes four values from the stack and swaps
                    their order as pairs.
```

```
op_ifdup          (n -- 0 | n n)
                  Duplicates the top value from the stack if it
                  is NOT zero. Leaves 0 unchanged.

op_depth          (--n)
                  Adds the number of items on the stack to the
                  stack.

op_drop           (n--)
                  Discards the top item from the stack.

op_dup            (n -- n n)
                  Duplicates the top item from the stack.

op_nip            (n n1 -- n1)
                  Discards the second item in the stack.

op_over           (n n1 -- n n1 n)
                  Copies the second item in the stack over the
                  top.

op_pick           (xu ... x0 u -- xu ... x0 xu)
                  Copies the u-th item on the stack to the top.

op_roll           (xu xu-1 ... x0 u -- xu-1 ... x0 xu)
                  Moves the u-th item on the stack to the top,
                  shifting items above it down.

op_rot            (n n1 n2 -- n1 n2 n)
                  The top three items on the stack are rotated
                  left, with the third item going to the top.

op_swap           (n n1 -- n1 n)
                  The top two items on the stack's order is
                  reversed.

op_tuck           (n n1 -- n1 n n1)
                  The top item on the stack is copied in front
                  of the second item.
```

Size & Separator OP Codes

```
op_size           (x -- x n)
                  Pushes the byte length of the top stack element
                  without consuming it.

op_codeseparator  (--)
                  Marks the beginning of signature-checked data.
                  All signature checks only use data after the
                  most recent OP\_CODESEPARATOR.

op_true           ( -- 1)
```

```
                    Pushes 1 onto the stack. Alias for OP\_1.
```

Splice Operator OP Codes

```
op_cat      (--)
            Concatenates the two strings provided.

op_split    (n n1--)
            Splits byte sequence n at n1.

op_num2bin  (n n1--)
            Converts value n into a byte sequence of length n1.

op_bin2num  (n--)
            Converts byte sequence n into a numerical value.

op_substr   (start length --) data: (data -- substring)
            Extracts a substring by start index and length.
            Chronicle upgrade (0xb3).

op_left     (n --) data: (data -- left)
            Extracts the leftmost N bytes.
            Chronicle upgrade (0xb4).

op_right    (n --) data: (data -- right)
            Extracts the rightmost N bytes.
            Chronicle upgrade (0xb5).
```

Bitwise Operator OP Codes

```
op_invert       (n -- n)
                Flips all the bits in the input.

op_and          (n1 n2 -- AND)
                Logical AND performed and the result added to
                the stack.

op_or           (n1 n2 -- OR)
                Logical OR performed and the result added to
                the stack.

op_xor          (n1 n2 -- XOR)
                Logical XOR performed and the result added to
                the stack.

op_equal        (n n1 -- f)
                Returns a flag depending on if n1 and n2 are
                equal. 1 if true, 0 if false.

op_equalverify  (n1 n2 -- f)
                OP_EQUAL then OP_VERIFY.
```

```
op_reserved1    (--)
                Renders transaction invalid unless in
                unexecuted OP_IF statement.

op_reserved2    (--)
                Renders transaction invalid unless in
                unexecuted OP_IF statement.
```

Arithmetic Operator OP Codes

```
op_1add                 (n -- n+1)
                        1 is added to the top value on the stack.

op_1sub                 (n -- n-1)
                        1 is subtracted from the top value.

op_2mul                 (n -- n*2)
                        The top value on the stack is doubled.

op_2div                 (n -- n/2)
                        The top value on the stack is halved.

op_negate               (n -- -n)
                        The top value's sign is flipped.

op_abs                  (n -- |n|)
                        The absolute value of the top item on
                        the stack is returned.

op_not                  (n -- f)
                        Logical NOT. Returns 1 if the input is 0,
                        otherwise returns 0.

op_0notequal            (n -- f)
                        Returns true if the value is not zero,
                        else false.

op_add                  (n n1 -- n3)
                        Top two items from the stack are summed
                        and the value returned.

op_sub                  (n n1 -- n3)
                        The top value from the stack is
                        subtracted from the second value.

op_mul                  (n n1 -- n3)
                        The top value is multiplied by the
                        second value in the stack.

op_div                  (n n1 -- n3)
                        The second value in the stack is divided
                        by the top value.
```

```
op_mod                  (n n1 -- n3)
                        The second value is divided by the top
                        value and the remainder returned.

op_lshift               (n n1 -- n3)
                        The second value is logical left shifted
                        by the top value.

op_rshift               (n n1 -- n3)
                        The second value is logical right shifted
                        by the top value.

op_lshiftnum            (n shift -- result)
                        Numeric left shift preserving sign.
 Distinct
                        from op\_lshift which is bitwise on bytes.
                        Chronicle upgrade (0xb6).

op_rshiftnum            (n shift -- result)
                        Numeric right shift preserving sign.
 Distinct
                        from op\_rshift which is bitwise on bytes.
                        Chronicle upgrade (0xb7).

op_booland              (n n1 -- f)
                        True flag returned if both n and n1 are
                        non-zero, else false.

op_boolor               (n n1 -- f)
                        True flag returned if either n or n1 is
                        not 0, else false.

op_numequal             (n n1 -- f)
                        True if n is equivalent to n1, else
                        false.

op_numequalverify       (n n1 -- f)
                        OP_NUMEQUAL then OP_VERIFY.

op_numnotequal          (n n1 -- f)
                        True if n is NOT equal to n1, else
                        false.

op_lessthan             (n n1 -- f)
                        True if n is less than n1, else false.

op_greaterthan          (n n1 -- f)
                        True if n is greater than n1, else
                        false.

op_lessthanorequal      (n n1 -- f)
                        True if n is less than or equal to n1,
                        else false.
```

```
op_greaterthanorequal  (n n1 -- f)
                       True if n is greater than or equal to
                       n1, else false.

op_min                 (n n1 -- n)
                       The top two values from the stack are
                       taken and the smaller returned.

op_max                 (n n1 -- n)
                       The top two values from the stack are
                       taken and the larger returned.

op_within              (n n1 n2 -- f)
                       True is returned if n is in between n1
                       and n2.
```

Crypto Operator OP Codes

```
op_ripemd160            (--)
op_sha1                 (--)
op_sha256               (--)
op_hash160              (--)
op_hash256              (--)
op_checksig             (--)
op_checksigverify       (--)
op_checkmultisig        (--)
op_checkmultisigverify  (--)
```

Lock Time Operator OP Codes

```
op_nop2           (--)
op_nop3           (--)
op_pubkeyhash     (--)
op_pubkey         (--)
op_invalidopcode  (--)
```

Reserved OP Codes

```
op_nop1    (--)
op_nop4    (--)
op_nop5    (--)
op_nop6    (--)
op_nop9    (--)
op_nop10   (--)
```

> **Chronicle Upgrade (Block 943,816):** NOP4–8 (0xb3–0xb7) were repurposed as op_substr, op_left, op_right, op_lshiftnum, and op_rshiftnum. The transaction version (op_ver) auto-switches from 1 to 2 at the activation block.

5

Henceforth includes a full Bitcoin SV (BSV) wallet, enabling users to create wallets, receive and send transactions, and interact with the blockchain — all from the same app that runs FORTH programs.

5.1 Wallet Architecture

The wallet is built on a hierarchical deterministic (HD) key structure, meaning a single seed phrase generates an unlimited number of addresses.

- **Seed Phrase:** A 12-word BIP-39 mnemonic is generated when creating a new wallet. This seed phrase is the master backup — from it, every key can be regenerated.
- **Key Derivation:** Two derivation methods are supported:
  - **Type42 (BRC-42)** — the default for new wallets. Uses elliptic curve key tweaking (private key addition, public key multiplication) for more flexible derivation paths.
  - **BIP-44** — the legacy standard. Retained for restoring wallets created by other software. Uses the standard path m/44'/236'/0'/0/n.
- **Address Rotation:** After each transaction, a new receiving address is derived automatically. This improves privacy by avoiding address reuse.

> **Security:** Private keys and seed phrases are stored exclusively in the iOS Keychain, protected by the device's Secure Enclave. They never leave the device and are never transmitted over the network.

5.1.1 Address Discovery

The wallet discovers addresses with funds using a shared Type42 scanner (scanType42-Addresses). The scanner is derivationType-independent — it always tries Type42 first if the master key exists in Keychain, regardless of what derivationType says on the wallet card. This prevents funds from becoming invisible if the derivation type metadata is wrong.

- **Dynamic scan range:** Scans at least 20 addresses (gap limit) past the last address with activity, with a minimum floor of 50 addresses.

- **Receive and change chains:** Scans both 1-wallet-{N} (receive) and 0-change-{N} (change) invoice numbers.
- **BIP-44 fallback:** After Type42, scans standard BIP-44 paths. A wallet can have funds on both derivation paths.
- **Auto-repair:** If Type42 addresses are discovered but derivationType is wrong, the scanner automatically corrects it and persists the fix.

5.2 Wallet Cards

Each wallet is represented by a Wallet Card in the app interface. A card stores:

- **Wallet title** — a user-chosen name
- **Main address** — the primary receiving address
- **Derived addresses** — additional addresses generated through key rotation
- **Balance** — confirmed and unconfirmed satoshis, reconciled from the UTXO set
- **Derivation type** — Type42 or BIP-44

Multiple wallets can be created and managed simultaneously. The wallet card interface supports creating, renaming, and deleting wallets, with biometric (Face ID / Touch ID) authentication required to access sensitive details.

Default Wallet

One wallet can be marked as the default wallet, indicated by a star on the wallet card. The default wallet is used by the terminal send word and other non-UI operations. If no default is set, the first wallet is used.

To change the default wallet, open the wallet details view and tap **Set as Default Wallet**. The setting persists across app launches.

5.3 Transactions

5.3.1 Receiving

To receive BSV, share your wallet's receiving address or scan its QR code. The wallet monitors for incoming transactions using:

1. **JungleBus REST polling** — polls every 30 seconds for new transactions via the GET /v1/address/get/{address} endpoint. When a new transaction ID is detected, the wallet triggers an automatic refresh.

2. **Periodic sync** — full balance and UTXO refresh as a safety net

When a new transaction is detected, the wallet automatically refreshes the balance, UTXOs, and transaction history.

5.3.2 Notifications

The wallet uses a UTXO-based deposit detection system. Instead of comparing balances (which can fluctuate mid-sync), it tracks known UTXO outpoints (txHash:txPos) per wallet in UserDefaults. Only genuinely new outpoints trigger a notification — the system is immune to confirmation state changes and mid-sync transient balances.

When a wallet is first seen, existing outpoints are seeded silently (no "received entire balance" notification on first install).

5.3.3 Sending

Transactions are built locally on the device:

1. **UTXO Selection:** The wallet selects unspent transaction outputs to fund the payment, avoiding ordinal-protected UTXOs.
2. **Fee Calculation:** Fees are calculated using the current rate from the preferred miner (GorillaPool or TAAL) via ARC policy endpoints.
3. **Change Output:** Any excess is returned to a fresh change address derived from the wallet.
4. **Signing:** Each input is signed with the correct private key for its address (Type42 or BIP-44 derived).
5. **Anti-Fee-Sniping:** The nLockTime field is set to the current block height, preventing miners from re-mining old transactions for higher fees.
6. **Broadcasting:** The signed transaction is submitted via ARC with automatic failover between GorillaPool and TAAL.

The pay word in the FORTH terminal opens the transaction builder. The pay view has three tabs for selecting what to broadcast:

- **OP_RETURN** — write a text message to embed on-chain
- **Script** — select a saved script or a .fs script file
- **Upload** — pick a photo, video, or file from the device and upload it on-chain via the B:// protocol

> **Smart Sync:** When a payment is initiated, UTXOs are loaded from the local disk cache first (instant). A network fetch only occurs on genuine first launch when no UTXO file exists. Fee quotes are cached for 5 minutes. A times-table circle animation (cardioid pattern) plays during broadcast. The payment flow trusts local UTXO state — if a UTXO

> turns out to be already spent, ARC will reject with DOUBLE_SPEND_ATTEMPTED.

After a successful broadcast, the wallet builds a local transaction model from the data it already has (inputs, outputs, addresses, amounts) and inserts it into the transaction history immediately — no network fetch required. The ARC-returned txid is verified against the locally computed txid as cryptographic proof that the network accepted exactly what was built. Spent UTXOs are removed from memory (preventing double-spend on rapid re-send) and change UTXOs are added. Full transaction details from WhatsOnChain overwrite the local entry on the next sync.

> **Trust Local State:** Per the Bitcoin white paper (§2), we built the transaction and signed it with our private keys — we know exactly what it contains. There is no need to ask a third party (WhatsOnChain) what we just sent. ARC confirms the network saw it; the rest we already know.

5.3.4 Paymail

Paymail allows sending BSV to human-readable addresses (e.g., user@example.com) instead of raw Bitcoin addresses. The wallet implements:

- **BRC-30 receive-transaction callbacks** — P2P payment destinations resolved via the .well-known/bsvalias capability discovery. The payee's server provides a fresh output script for each payment.
- **Automatic resolution** — when a paymail address is entered in the pay view, the wallet resolves it to a Bitcoin address transparently.

5.3.5 Transaction History

The wallet displays a chronological transaction history with:

- **Direction detection** — incoming (received) vs outgoing (sent), determined by checking whether any inputs belong to the wallet
- **Grouping** — transactions are grouped by: Pending, Today, Yesterday, This Week, This Month, Earlier
- **Search and filter** — filter by amount, date range, or search by transaction ID
- **SPV verification** — transactions can be verified against block headers using Simplified Payment Verification

5.4 UTXOs and Balance

A UTXO (Unspent Transaction Output) represents a discrete amount of Bitcoin that can be spent. The wallet tracks all UTXOs across all addresses.

Balance from UTXOs: The displayed balance is computed solely from the UTXO set — no separate balance API call is needed. When UTXOs are fetched, the balance is calculated as:

```
confirmed   = sum of UTXOs with block height > 0
unconfirmed = sum of UTXOs with block height = 0
total       = confirmed + unconfirmed
```

Satoshi conversions use the named constant satoshisPerBSV (100,000,000) and price conversions use priceCentsDivisor (100.0). Raw magic numbers are never used.

5.4.1 Ordinal Protection

1Sat Ordinals are NFTs inscribed on single-satoshi UTXOs. The wallet protects ordinals at two levels:

1. **Transaction builder:** UTXOs with a value of 1 satoshi are excluded from UTXO selection during transaction building — a fast local check with no network call.
2. **Inscription scanner:** For display in the Ordinals view, 1-sat UTXOs are scanned by fetching the raw transaction and detecting the OP_FALSE OP_IF ... OP_ENDIF inscription pattern. Non-1-sat UTXOs are skipped entirely (ordinals are always exactly 1 sat). Results are cached per outpoint — once a UTXO has been scanned, it is never re-fetched from the network.

5.4.2 UTXO Split

From the UTXO detail view, a single UTXO can be split into up to 240 separate outputs across fresh derived addresses. This is useful for:

- Preparing UTXOs for future transactions (avoiding the need to wait for change confirmations)
- Distributing funds across addresses for privacy
- Creating multiple 1-sat UTXOs for ordinal inscriptions

5.5 API Providers

The wallet uses a hybrid architecture with multiple blockchain service providers:

Capability	WhatsOnChain	GorillaPool	TAAL	JungleBus
UTXOs (balance derived)	✓	✗	✗	✗
Transaction history	✓	✗	✗	✓
SPV Merkle proofs	✓	✗	✗	✗
BSV price	✓	✗	✗	✗
Fee quotes	✗	✓	✓	✗
Broadcast (ARC)	✗	✓	✓*	✗
Real-time monitoring	✗	✗	✗	✓

*TAAL requires a user-configured API key. GorillaPool is the primary broadcaster; TAAL is failover.

Each provider handles what it does best. API keys and subscription IDs are stored securely in the iOS Keychain and can be configured in Settings → API & Miners.

5.6 Security

- **Biometric authentication** — Face ID or Touch ID required to view private keys, seed phrases, and wallet details. Sessions expire after 120 seconds of inactivity.
- **Keychain storage** — All sensitive material (keys, seeds, API tokens) stored with .whenUnlockedThisDeviceOnly accessibility — encrypted at rest, tied to the device.
- **Seed verification** — After creating a wallet, users are prompted to verify their seed phrase via a 3-word quiz.
- **No server dependency** — The wallet is fully self-custodial. No account, no registration, no server holds your keys.
- **Message encryption (BRC-2)** — Electrum ECIES format for end-to-end encrypted messages between BSV keys. Uses ECDH key agreement, AES-256-CBC encryption, and HMAC-SHA256 authentication. Wire-compatible with the @bsv/sdk EncryptedMessage class. Format: "BIE1" ‖ pubkey ‖ ciphertext ‖ HMAC.
- **Wallet file encryption** — Wallet data files encrypted with AES-256-GCM using a Keychain-backed key before syncing to iCloud.

5.6.1 QR Code Scanner

A unified QR scanner in Settings handles three QR code types, routed automatically by content:

1. **hfauth: prefix** — BRC-103 authentication flow (see below)
2. **WIF private key** (Base58Check version byte 0x80) — prompts for a wallet name, then imports via Wallet.restoreFromWIF()
3. **Bitcoin address** (version byte 0x00) — opens the pay view pre-filled with the scanned address

5.6.2 BRC-103 Authentication

BRC-103 provides passwordless authentication to web services using Bitcoin key pairs:

1. A web service displays a QR code containing a challenge and callback URL (prefixed with hfauth:)
2. The wallet scans the QR, displays the requesting domain, and asks for user approval

3. On approval, the wallet signs the challenge with the wallet's identity key and sends the signature to the callback URL
4. The web service verifies the signature against the wallet's public key

No passwords, no accounts — just cryptographic proof of key ownership.

5.6.3 BRC-100 Wallet Interface

The wallet exposes a subset of its capabilities via a URL scheme for integration with external applications:

```
henceforth://wallet/{method}?params=...
```

Supported operations include key derivation, signing, encryption, and certificate management (BRC-52/53). All sensitive operations require user approval via an on-screen confirmation sheet with biometric authentication.

The interface follows the BRC-100 specification: a WalletInterface protocol defines the contract, and a HenceforthWallet adapter bridges to the existing wallet, FORTH interpreter, and ARC broadcasting infrastructure.