Network Working Group
Internet-Draft
Intended status: Informational
Expires: 23 April 2026

H. O'Connor
Anuna Research Pty Ltd

20 October 2025

SayWhere Geocoding System: Human-Memorable Geographic Coordinates
draft-saywhere-geocoding-01

Abstract

This document specifies SayWhere, an open-source system for encoding geographic coordinates (latitude, longitude, and optional altitude) into human-memorable word phrases and decoding them back to coordinates. The system provides deterministic, reversible encoding with two-layer error detection (per-word parity and optional terminal word-phrase checksum) and supports three-dimensional positioning for multi-level structures. SayWhere introduces hierarchical variable-length phrases with true prefix relationships, enabling precision scaling from regional (1 word) to meter-level accuracy (6 words). The system uses the Bitcoin Improvement Proposal 39 (BIP-39) mnemonic wordlist for multilingual support and geohash spatial indexing for efficient location encoding.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 23 April 2026.

Copyright Notice

Table of Contents

1. Introduction

Geographic coordinates (latitude, longitude) are precise but difficult to communicate verbally or memorize. A typical coordinate pair such as (40.7128, -74.0060) requires exact decimal precision and is prone to transcription errors. SayWhere addresses this by encoding coordinates into memorable word phrases while maintaining precision, determinism, and providing error detection capabilities.

This challenge becomes critical in natural disasters and emergency response scenarios, where communication infrastructure may be compromised and rapid verbal location sharing essential for coordinating rescue operations. Word-based location phrases can be communicated over radio, written on paper, or relayed through human chains without requiring functional GPS devices or internet connectivity.

Traditional postal addresses do not always provide the level of precision desired, assume a built environment, and often require some tacit knowledge of that built environment. Existing proprietary systems use closed algorithms, lack transparency and require a centralized service to resolve word phrases creating a single point of failure which is unaceptable in critical environments. Plus Codes use alphanumeric strings that are harder to communicate verbally. Traditional geohash uses Base32 strings that are not human-memorable. SayWhere provides an open, transparent alternative with hierarchical structure, optional altitude, multi-lingual support, and error detection.

The design goals for SayWhere are:

1. *Deterministic*: Same coordinates always produce identical word phrases
2. *Reversible*: Word phrases can be decoded back to original coordinates
3. *Human-Friendly*: Words are common, easy to spell, and phonetically distinct
4. *Error Detection*: Two-layer validation with per-word parity and optional terminal checksum tp validate a word phrase.
5. *3D Support*: Optional altitude component for vertical positioning
6. *Open Standard*: Public domain algorithm based on established prior art with no proprietary restrictions

7. *Hierarchical*: Variable-length phrases (1-6 words) with true prefix relationships

Compared to existing systems:

* *Proprietary word-based systems*: Closed algorithms, fixed-length (typically 3 words), no error detection, no altitude support

* *Plus Codes*: Alphanumeric (not word-based), harder to communicate verbally

* *Geohash*: Base32 strings, not human-memorable

* *SayWhere*: Open source, hierarchical word phrases, 3D support, two-layer error detection (LSB parity + optional terminal checksum), true prefix relationships, hand-verifiable checksums

SayWhere is built upon established open technologies as prior art:

* *Geohash* (Niemeyer, 2008): Spatial indexing system using base-32 strings

* *BIP-39* (Palatinus et al., 2013): Mnemonic wordlist standard with 2,048 words

SayWhere combines geohash spatial indexing with BIP-39 word encoding to create hierarchical variable-length phrases. This differs from fixed-length proprietary systems that use cell-based integer decomposition approaches. The use of geohash as the intermediate representation (rather than proprietary integer encoding) and the hierarchical prefix relationships are key distinguishing features.

2. Terminology

2.1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

2.2. Core Concepts

Hierarchical Word Phrase: Variable-length sequence of words (1-N words) representing location with increasing precision

Prefix Relationship: Shorter phrases represent a bounded area that

contain all longer phrases with the same prefix

Location Words: Words encoding the position of cell within a base32 cell grid (excluding checksum and altitude)

Checksum Word: Optional terminal validation word from a distinct 32-word vocabulary for phrase-level error detection

Altitude Units: Optional numeric component for vertical position

Wordlist: An ordered array of words used for encoding/decoding (BIP-39 standard)

Precision Level: Spatial resolution determined by phrase length

Geohash: A geocoding system using base32 strings for spatial indexing

2.3. Phrase Types

Hierarchical Phrase: Pure spatial encoding without terminal checksum (maintains prefix relationships)

Validated Phrase: Includes terminal checksum word for phrase-level error detection

3D Phrase: Includes altitude component for vertical positioning

3. System Architecture

3.1. Encoding Pipeline

The encoding process transforms geographic coordinates into word phrases through the following pipeline:

1. Input validation: Verify latitude [-90, 90], longitude [-180, 180], altitude [-30000, 30000]

2. Geohash generation: Convert coordinates to base32 geohash string

3. Chunk processing: Split geohash into 2-character chunks (10 bits each)

4. Parity calculation: Add 1-bit LSB parity to each chunk

5. Word mapping: Map 11-bit values (10 data + 1 parity) to BIP-39 wordlist

6. Phrase assembly: Join location words with dots for hierarchical structure

7. Optional: Compute and append CRC-8 terminal checksum word from 32-word vocabulary

3.2. Decoding Pipeline

The decoding process reverses the encoding to recover coordinates:

1. Phrase parsing: Split on dots, validate word format

2. Checksum detection: Identify and separate terminal checksum word (if present)

3. Word lookup: Find each location word in BIP-39 wordlist to get 11-bit index

4. LSB parity validation: Verify 1-bit parity for each location word

5. Terminal checksum validation: If present, validate phrase-level checksum

6. Chunk extraction: Extract 10-bit chunk value from each location word

7. Geohash reconstruction: Concatenate chunks to form geohash string

8. Coordinate decoding: Decode geohash to latitude/longitude with bounds

3.3. Format Specifications

3.3.1. Hierarchical Format

Hierarchical phrases maintain true prefix relationships:

```
grape                                 # Region (~900km)
grape.column                          # City (~28km)
grape.column.hip                      # Neighborhood (~900m)
grape.column.hip.thought              # Street (~27m)
grape.column.hip.thought.pull         # Door (~90cm)
grape.column.hip.thought.pull.wave    # Sub-meter (~2.7cm)
```

Key Property: Each shorter phrase represents an area containing all longer phrases with the same prefix.

Error Detection: Two independent layers: 1. Each location word contains a built-in LSB parity bit for single-word error detection 2. Optional terminal checksum word validates the entire phrase (with ~3% False Negative)

With Terminal Checksum: ~~~~ grape.column.hip.seal # Validated 3-word phrase ~~~~ Where "seal" is from the 32-word checksum vocabulary and validates "grape.column.hip".

3.3.2. Precision vs. Length Table

Precision is constrained by geohash spatial indexing:

Words	Geohash Chars	Grid Dimensions	Avg Size	Use Case
1	2	1,252km × 624km	~900km	Country/ Large Region
2	4	39.1km × 19.5km	~28km	City
3	6	1.2km × 609m	~900m	Neighborhood
4	8	38.2m × 19m	~27m	Building
5	10	1.2m × 59.5cm	~90cm	Room/Precise Position
6	12	3.7cm × 1.9cm	~2.7cm	Sub-Meter Precision
7	14	1.2mm × 0.6mm	~0.85mm	Millimeter Precision
8	16	37μm × 19μm	~27μm	Micron-Level Precision

Table 1: Precision Mapping for Hierarchical Phrases

Key Insight: The 2,048-word BIP-39 vocabulary provides sufficient combinations for encoding geohash data. Each word encodes 2 geohash characters (10 bits) plus 1 parity bit, utilizing the full 11-bit address space.

Extensibility: The hierarchical encoding pattern can be extended beyond 6 words as measurement devices and positioning technologies improve. The algorithm supports phrases of any length, with each additional word adding approximately 2 geohash characters of precision. Future applications with centimeter or millimeter-accurate positioning systems can use 7, 8, or more words while maintaining full backward compatibility with shorter phrases.

4. Wordlist Specification

4.1. BIP-39 Wordlist (REQUIRED)

SayWhere uses the BIP-39 mnemonic wordlist [BIP39] as its standard wordlist. This provides significant advantages:

* *Size:* Exactly 2,048 words (11 bits) - perfect for encoding 2 geohash characters (10 bits) + 1 parity bit

* *Battle-tested:* Used by millions in cryptocurrency wallets since 2013

* *Multilingual:* Official wordlists in 9 languages

* *Widely distributed:* BIP-39 wordlists are embedded in countless applications and hardware devices worldwide, ensuring availability and consistency

* *Open source:* No licensing restrictions

* *Existing ecosystem:* Libraries available in all major languages

4.2. Requirements

A compliant wordlist MUST satisfy these criteria:

1. *Size:* Exactly 2,048 words (index 0 to 2,047)

2. *Character Set:* Lowercase ASCII letters [a-z] only

3. *Length:* Each word 3-8 characters (BIP-39 standard)

4. *Uniqueness:* No duplicate words

5. *Ordering:* Stable ordering (indices never change within a version)

6. *Phonetic Distinctness:* First 4 letters must be unique (BIP-39 property)

7. *Family-Friendly:* No profanity or offensive terms

4.3. File Format

Wordlists MUST be stored in JSON format:

```
{
  "version": "1.0.0",
  "language": "en",
  "description": "BIP-39 English wordlist for SayWhere",
  "created": "2025-10-10",
  "word_count": 2048,
  "source": "https://github.com/bitcoin/bips/blob/master/bip-0039/english.txt",
  "license": "MIT",
  "words": [
    "abandon",
    "ability",
    "able",
    "...",
    "zoo"
  ]
}
```

4.4. Available Wordlists

BIP-39 provides wordlists in the following languages:

* English
* Spanish
* French
* Italian
* Japanese
* Korean
* Chinese (Simplified)
* Chinese (Traditional)
* Czech
* Portuguese

4.5. Versioning

Wordlist versions MUST follow semantic versioning (MAJOR.MINOR.PATCH):

* *MAJOR:* Incompatible changes (words removed/reordered)

* *MINOR:* Backward-compatible additions

* *PATCH:* Bug fixes (typo corrections)

Critical: Encoding and decoding MUST use the same wordlist version.

5. Encoding Algorithm

5.1. Hierarchical Encoding

5.1.1. Input Parameters

```
latitude:     float  # -90.0 to 90.0 (degrees)
longitude:    float  # -180.0 to 180.0 (degrees)
altitude:     float  # OPTIONAL, in meters (planned)
max_precision: int   # 1-6 words (default: 3)
wordlist:     array  # Array of 2,048 words (BIP-39)
```

5.1.2. Algorithm Steps

Step 1: Validate Input

```
IF latitude < -90.0 OR latitude > 90.0:
    RAISE ERROR "Invalid latitude"

IF longitude < -180.0 OR longitude > 180.0:
    RAISE ERROR "Invalid longitude"

IF max_precision < 1 OR max_precision > 6:
    RAISE ERROR "max_precision must be 1-6"

IF length(wordlist) != 2048:
    RAISE ERROR "Invalid wordlist size"
```

Step 2: Generate Full Geohash

```
# Each word encodes 2 geohash characters
geohash_chars = max_precision * 2
geohash = geohash_encode(latitude, longitude, geohash_chars)

# Example: (40.7128, -74.0060, 3 words) → 6 chars → "dr5reg"
```

Step 3: Chunk-Based Word Generation

```
FUNCTION geohash_to_word_indices_hierarchical(geohash, word_count):
    """
    Convert geohash to word indices using chunk-based encoding.

    KEY PROPERTIES:
    1. Each word encodes 2 geohash characters (10 bits)
    2. BIP-39 wordlist (2048 words = 11 bits) provides 1 extra bit for checksum
    3. Fully reversible: words → geohash is deterministic O(1) operation
    4. Prefix-preserving: shorter phrases are prefixes of longer ones
    """
    base32_alphabet = "0123456789bcdefghjkmnpqrstuvwxyz"
    wordlist_size = 2048  # BIP-39 standard
    word_indices = []

    # Each word encodes a 2-character geohash chunk
    FOR position FROM 0 TO word_count - 1:
        # Extract 2-character chunk at this position
        chunk_start = position * 2
        chunk_end = chunk_start + 2
        chunk = geohash[chunk_start : chunk_end]

        # Convert 2-char chunk to 10-bit value (0-1023)
        char1 = chunk[0]
        char2 = chunk[1]
        char1_index = index_of(char1 in base32_alphabet)  # 0-31 (5 bits)
        char2_index = index_of(char2 in base32_alphabet)  # 0-31 (5 bits)

        chunk_value = (char1_index * 32) + char2_index  # 0-1023 (10 bits)

        # Calculate simple parity checksum (1 bit)
        checksum_bit = popcount(chunk_value) MOD 2  # Count of 1-bits mod 2

        # Combine: (10-bit data << 1) | 1-bit checksum = 11-bit word index
        word_index = (chunk_value << 1) | checksum_bit  # 0-2047

        word_indices.append(word_index)

    RETURN word_indices
END FUNCTION
```

Step 4: Build Hierarchical Phrases

```
phrases = []

FOR word_count FROM 1 TO max_precision:
    # Geohash prefix for this precision (2 chars per word)
    geohash_prefix = geohash[0 : word_count * 2]

    # Generate word indices for this precision level
    word_indices = geohash_to_word_indices_hierarchical(geohash_prefix, word_count)

    # Map indices to words
    words = [wordlist[idx] for idx in word_indices]

    # Join with dots
    phrase = join(words, ".")
    phrases.append(phrase)

RETURN phrases
```

5.1.3. Example Execution

Input: New York City (40.7128, -74.0060), max_precision = 3

```
Geohash (6 chars): "dr5reg"

# Word 1: Encodes "dr" (chars 0-1)
chunk_1 = "dr"
char1_index = 12 (d), char2_index = 23 (r)
chunk_value_1 = 12 * 32 + 23 = 407
checksum_1 = popcount(407) % 2 = 6 % 2 = 0
word_index_1 = (407 << 1) | 0 = 814
→ wordlist[814] = "grape"

Phrase 1: "grape"  # ~900km region

# Word 2: Encodes "5r" (chars 2-3)
chunk_2 = "5r"
char1_index = 5 (5), char2_index = 23 (r)
chunk_value_2 = 5 * 32 + 23 = 183
checksum_2 = popcount(183) % 2 = 6 % 2 = 0
word_index_2 = (183 << 1) | 0 = 366
→ wordlist[366] = "column"

Phrase 2: "grape.column"  # ~28km city

# Word 3: Encodes "eg" (chars 4-5)
chunk_3 = "eg"
char1_index = 13 (e), char2_index = 15 (g)
chunk_value_3 = 13 * 32 + 15 = 431
checksum_3 = popcount(431) % 2 = 7 % 2 = 1
word_index_3 = (431 << 1) | 1 = 863
→ wordlist[863] = "hip"

Phrase 3: "grape.column.hip"  # ~900m neighborhood

Output:
[
    "grape",                  # ~900km region
    "grape.column",           # ~28km city
    "grape.column.hip"        # ~900m neighborhood
]
```

Prefix Guarantee: Each phrase is a proper prefix of all subsequent phrases. This ensures true spatial hierarchy where shorter phrases represent areas that contain all locations with longer matching prefixes.

6. Decoding Algorithm

6.1. Hierarchical Decoding (RECOMMENDED)

6.1.1. Input Parameters

```
   phrase:   string  # "word1.word2.word3..." (variable length)
   wordlist: array   # BIP-39 wordlist (2,048 words)
```

6.1.2. Algorithm Steps

Step 1: Parse Phrase

```
words = split(phrase, ".")
word_count = length(words)

IF word_count < 1 OR word_count > 6:
    RAISE ERROR "Invalid hierarchical phrase length (must be 1-6 words)"

FOR each word IN words:
    IF NOT is_lowercase_alphabetic(word):
        RAISE ERROR "Invalid word format"
    IF length(word) < 3 OR length(word) > 8:
        RAISE ERROR "Word length out of BIP-39 range"
```

Step 2: Decode Each Word with Checksum Validation

```
 FUNCTION decode_word_with_checksum(word, wordlist):
     """
     Decode a single word and validate its built-in checksum.
     Returns the 2-character geohash chunk.
     """
     base32_alphabet = "0123456789bcdefghjkmnpqrstuvwxyz"

     # Find word in wordlist
     word_index = index_of(word in wordlist)
     IF word_index == -1:
         RAISE ERROR "Word not found in wordlist"

     # Extract checksum bit (LSB)
     checksum_bit = word_index AND 1

     # Extract chunk value (10 bits)
     chunk_value = word_index >> 1  # 0-1023

     # Validate checksum
     expected_checksum = popcount(chunk_value) MOD 2
     IF checksum_bit != expected_checksum:
         RAISE ERROR "Checksum validation failed for word: " + word

     # Convert chunk value back to 2 geohash characters
     char1_index = chunk_value / 32  # Integer division
     char2_index = chunk_value MOD 32

     chunk = base32_alphabet[char1_index] + base32_alphabet[char2_index]

     RETURN chunk
 END FUNCTION

   *Step 3: Reconstruct Geohash*

   geohash = ""

   FOR each word IN words:
       chunk = decode_word_with_checksum(word, wordlist)
       geohash = geohash + chunk

   # Example: ["grape", "column", "hip"] → "dr" + "5r" + "eg" → "dr5reg"

   *Step 4: Decode Geohash to Coordinates*

   (latitude, longitude, lat_error, lon_error) = geohash_decode(geohash)

   # Standard geohash decoding algorithm
   # Returns center point and error bounds
```

Step 5: Return Result

```
RETURN {
    "coordinates": {
        "latitude": latitude,
        "longitude": longitude
    },
    "bounds": {
        "latitude": {"min": latitude - lat_error, "max": latitude + lat_error},
        "longitude": {"min": longitude - lon_error, "max": longitude + lon_error}
    },
    "precision_level": word_count,
    "geohash": geohash
}
```

7. Hierarchical Properties

7.1. Prefix Relationships

SayWhere hierarchical phrases exhibit true spatial prefix relationships:

```
grape                          # Contains all locations starting with "grape.*"
grape.column                   # Contains all locations starting with "grape.column.*"
grape.column.hip               # Contains all locations starting with "grape.column.hip.
*"
grape.column.hip.thought       # Contains all locations starting with "grape.column.hip.
thought.*"
```

Mathematical Property: If phrase A is a prefix of phrase B, then the spatial area of A contains the spatial area of B.

7.2. Spatial Containment

Containment Guarantee: Any location encoded as "grape.column.hip.thought.pull" will always be contained within the spatial bounds of:

* "grape.column.hip.thought" (more precise)
* "grape.column.hip" (less precise)
* "grape.column" (even less precise)
* "grape" (least precise)

7.3. Use Cases

7.3.1. Progressive Disclosure

```
"Meet me in tokyo"                    # City-level
"Meet me in shibuya.tokyo"           # District-level
"Meet me at station.shibuya.tokyo"   # Building-level
```

7.3.2. Search and Filtering

```
-- Find all locations in a neighborhood
SELECT * FROM locations WHERE phrase LIKE 'downtown.seattle.%'

-- Find all locations in a city
SELECT * FROM locations WHERE phrase LIKE 'seattle.%'
```

8. Checksum Computation

8.1. Design Philosophy

SayWhere uses per-word parity checksums for error detection. Each word contains a built-in 1-bit parity checksum in its LSB. This provides error detection while maintaining hierarchical prefix relationships.

8.2. Per-Word Parity Checksum

8.2.1. Purpose

Built-in per-word validation that:

1. Detects single-bit errors in each word independently
2. Maintains hierarchical prefix relationships
3. Requires no additional checksum words
4. Works for phrases of any length (1-6 words)

8.2.2. Algorithm

Each word in the BIP-39 wordlist (2,048 words = 11 bits) encodes:

* *10 bits:* Geohash chunk data (2 base32 characters)
* *1 bit:* Parity checksum (LSB)

```
FUNCTION compute_word_with_parity(chunk_value):
    """
    Encode a 10-bit geohash chunk into an 11-bit word index with parity.

    Args:
        chunk_value: Integer 0-1023 (10 bits of geohash data)

    Returns:
        word_index: Integer 0-2047 (11 bits: 10 data + 1 parity)
    """
    # Calculate even parity bit
    parity_bit = popcount(chunk_value) MOD 2

    # Combine: shift data left 1 bit, OR with parity in LSB
    word_index = (chunk_value << 1) | parity_bit

    RETURN word_index
END FUNCTION

FUNCTION validate_word_parity(word_index):
    """
    Validate the parity checksum of a word.

    Args:
        word_index: Integer 0-2047 from wordlist lookup

    Returns:
        Boolean: True if parity is valid, False otherwise
    """
    # Extract parity bit (LSB)
    stored_parity = word_index AND 1

    # Extract data bits
    chunk_value = word_index >> 1

    # Recalculate expected parity
    expected_parity = popcount(chunk_value) MOD 2

    RETURN stored_parity == expected_parity
END FUNCTION
```

8.2.3. Properties

- *Error Detection:* Detects any odd number of bit flips in the 10-bit chunk
- *Probability:* 50% chance of detecting even number of bit flips

* *No False Positives:* Valid words always pass parity check

* *Zero Overhead:* No additional words required

* *Prefix Preserving:* Shorter phrases remain valid prefixes

8.3. Optional Terminal Checksum

8.3.1. Purpose

An optional terminal checksum word MAY be appended to any phrase to provide phrase-level error detection for transmission errors. The terminal checksum:

1. Uses a distinct 32-word vocabulary (auto-detectable)

2. Validates the entire phrase (word substitution, omission, transposition)

3. Provides near-uniform distribution for optimal error detection

4. Complements per-word LSB parity for two-layer error detection

8.3.2. Checksum Wordlist

The terminal checksum uses a 32-word vocabulary distinct from the BIP-39 location wordlist:

```
# Colors (indices 0-15)
red, blue, green, yellow, orange, purple, pink, brown,
black, white, gray, silver, gold, bronze, cyan, magenta

# Animals (indices 16-31)
cat, dog, fox, bear, lion, wolf, eagle, hawk,
deer, fish, frog, snake, owl, crow, seal, whale
```

These words are: - *Distinct*: Never overlap with BIP-39 location words - *Memorable*: Colors and animals are universally understood - *Phonetically clear*: Easy to communicate verbally over radio/phone

8.3.3. CRC-8 Algorithm

For systems with computational capability (phones, computers, calculators), use CRC-8 for optimal error detection:

```
FUNCTION compute_terminal_checksum_crc8(location_words, wordlist):
    """
    Compute CRC-8 based terminal checksum.
    Provides uniform distribution across all 32 checksum words.

    Uses CRC-8-CCITT: polynomial 0x07, initial value 0xFF.
    Computes CRC over 11-bit word indices for optimal uniformity.
    """
    # Step 1: Get word indices from wordlist
    indices = []
    FOR word IN location_words:
        index = FIND_INDEX(word IN wordlist)  # 0-2047 (11 bits)
        indices.APPEND(index)
    END FOR

    # Step 2: Pack 11-bit indices into byte array
    num_bits = LENGTH(indices) * 11
    num_bytes = CEILING(num_bits / 8)
    bytes = ALLOCATE_BYTES(num_bytes)

    bit_position = 0
    FOR index IN indices:
        # Write 11 bits of index into byte array
        FOR bit FROM 0 TO 10:
            bit_value = (index >> (10 - bit)) AND 1
            byte_index = bit_position / 8  # Integer division
            bit_in_byte = bit_position MOD 8
            bytes[byte_index] |= (bit_value << (7 - bit_in_byte))
            bit_position = bit_position + 1
        END FOR
    END FOR

    # Step 3: Compute CRC-8 using lookup table
    crc = 0xFF  # Initial value
    FOR byte IN bytes:
        index = crc XOR byte
        crc = CRC8_TABLE[index]
    END FOR

    # Step 4: Map to checksum word
    checksum_index = crc MOD 32

    RETURN CHECKSUM_32[checksum_index]
END FUNCTION
```

The CRC-8 lookup table (256 bytes) is provided in the reference implementation.

Properties: - *Distribution:* Chi-square = 18.85 (essentially perfect uniform distribution) - *Error Detection:* 96.9% (theoretical maximum) - *Computation:* Fast with lookup table (~10 operations per byte) - *Implementation:* Available in all programming languages

Key Insight: Computing CRC over word indices rather than strings eliminates structural biases from dots and letter frequencies, achieving near-perfect uniformity (chi-square of 18.85 vs target of 44.3 for uniform distribution).

8.3.4. Examples

Example 1: New York City ~~~~ Location: "grape.column.hip"

CRC-8 Calculation: Word indices: [814, 366, 863] Packed bytes: [0xCC, 0xDB, 0x6F] CRC-8 (polynomial 0x07, init 0xFF): 0x1E Checksum index: 0x1E % 32 = 30 Checksum: CHECKSUM_32[30] = "seal"

Result: "grape.column.hip.seal" ~~~~

Example 2: London ~~~~ Location: "kit.puzzle"

CRC-8 Calculation: Word indices: [1004, 1434] Packed bytes: [0xFA, 0xD5, 0x00] CRC-8 (polynomial 0x07, init 0xFF): 0xA4 Checksum index: 0xA4 % 32 = 4 Checksum: CHECKSUM_32[4] = "orange"

Result: "kit.puzzle.orange" ~~~~

8.3.5. Encoding with Terminal Checksum

To create a validated phrase:

```
FUNCTION encode_with_checksum(latitude, longitude, num_words):
    # Standard hierarchical encoding
    location_words = encode_location(latitude, longitude, num_words)

    # Compute terminal checksum
    checksum_word = compute_terminal_checksum(location_words)

    # Append checksum
    RETURN location_words + [checksum_word]
END FUNCTION
```

8.3.6. Decoding with Checksum Detection

Decoders MUST automatically detect and validate terminal checksums:

```
   FUNCTION decode_with_checksum_detection(phrase):
       words = SPLIT(phrase, ".")

       IF LENGTH(words) < 2:
           # No checksum possible
           RETURN decode_location(words), checksum_valid=FALSE
       END IF

       last_word = words[LENGTH(words) - 1]

       # Check if last word is from checksum vocabulary
       IF last_word IN CHECKSUM_32:
           # Treat as checksum
           location_words = words[0 : LENGTH(words) - 1]
           provided_checksum = last_word

           # Compute expected checksum
           expected_checksum = compute_terminal_checksum(location_words)

           # Validate
           IF provided_checksum == expected_checksum:
               coordinates = decode_location(location_words)
               RETURN coordinates, checksum_valid=TRUE
           ELSE:
               RAISE ERROR "Terminal checksum validation failed"
           END IF
       ELSE:
           # All words are location words
           coordinates = decode_location(words)
           RETURN coordinates, checksum_valid=FALSE
       END IF
   END FUNCTION
```

8.3.7. Error Detection Capabilities

The terminal checksum detects:

* *Word substitution*: "grape" misheard as "grace"

* *Word omission*: "grape.column.hip" → "grape.hip"

* *Word transposition*: "grape.column" → "column.grape"

* *Word addition*: Unintended extra words

Detection Rate: 5-bit checksum provides ~97% detection of transmission errors (1/32 false negative rate).

8.3.8. Two-Layer Error Detection

Terminal checksum and per-word LSB parity work together:

Layer	Scope	Detects	When Checked
LSB Parity	Per-word	Bit corruption in individual words	During word decode
Terminal Checksum	Whole phrase	Wrong word, order, omissions	After all words decoded

Table 2

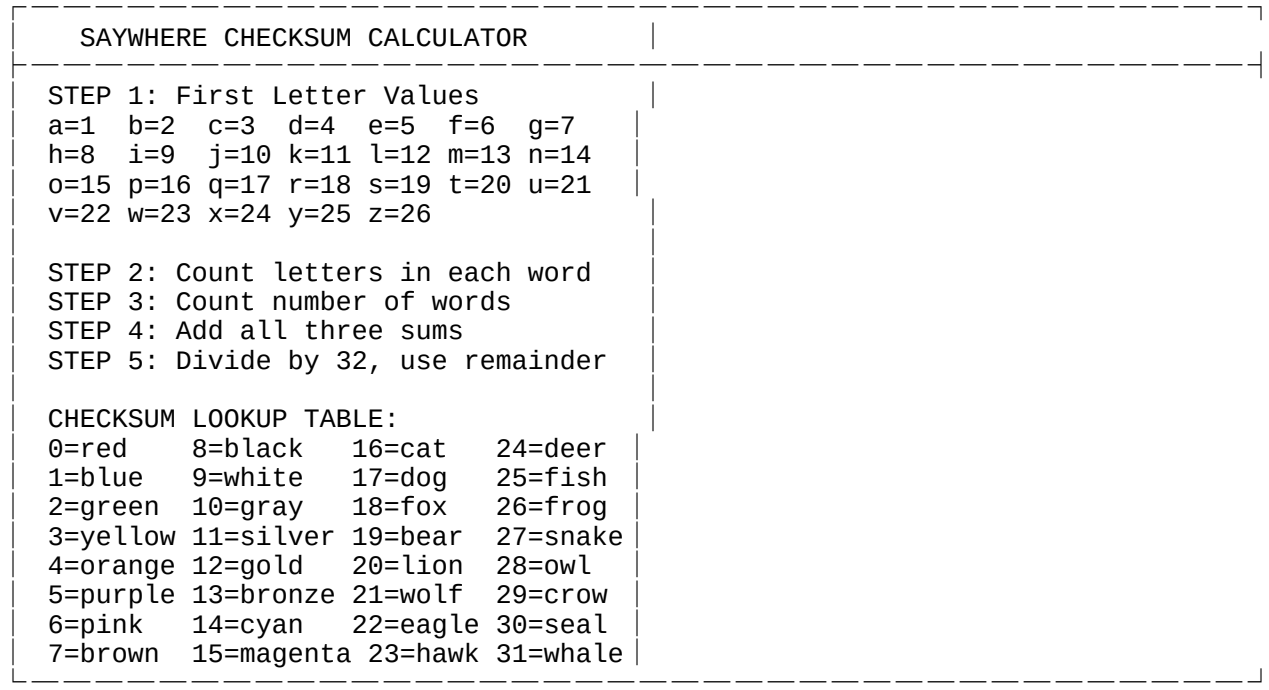
Example: Error caught by terminal checksum ~~~~ Sent: "grape.column.hip.seal" Received: "grape.color.hip.seal" (column → color)

9. LSB parity passes (both valid BIP-39 words)

```
# Terminal checksum fails: expected =
compute_terminal_checksum(["grape", "color", "hip"]) → "whale"
received = "seal" # Error detected!  ~~~~
```

9.1. Field Verification (BACKUP)

The following simplified checksum MAY be calculated and verified in emergency situations using printed reference cards:



9.1.1. Use Cases

Emergency Response ~~~~ Hiker (over radio): "My location is grape column hip seal" Dispatcher: _validates checksum_ System: Checksum valid - location confirmed Dispatcher: "Coordinates confirmed, dispatching rescue team" ~~~~

Error Detection ~~~~ Sent: "grape.column.hip.seal" Received: "grape.hip.seal" (word dropped)

System calculates: Expected: compute_terminal_checksum(["grape", "hip"]) → "yellow" Received: "seal" Mismatch - transmission error detected

Response: "Please repeat location, checksum error detected" ~~~~

10. Altitude Support

10.1. Design Philosophy

SayWhere altitude components are expressed as unitless integers representing quantized steps. The step size is a configuration parameter, not part of the phrase itself.

Rationale: Unitless values maintain:

* *Phrase brevity:* ".20" vs ".60m" or ".20meters"

* *Simplicity:* Consistent with horizontal coordinate abstraction

* *Flexibility:* Applications can choose appropriate step sizes without breaking compatibility

* *Voice-friendly:* Easier to communicate "twenty" than "twenty meters"

10.2. Quantization

Altitude MUST be quantized to fixed-step increments:

```
altitude_step = 3.0  # meters (configuration parameter, not in phrase)
altitude_units = ROUND(altitude_meters / altitude_step)
```

Standard step configurations:

Use Case	Step Size	Example Input	Encoded Units	Notes
Buildings	3.0m	60.0m	.20	Typical floor height
Drones	1.0m	60.0m	.60	Precise vertical control
Aircraft	100.0m	6000.0m	.60	Flight levels (FL060)
Mining/ Caves	5.0m	-50.0m	.-10	Underground operations

Table 3: Altitude Step Configurations

10.3. Range Limits

```
MIN_ALTITUDE_UNITS = -1000  # Lowest representable value
MAX_ALTITUDE_UNITS = 10000  # Highest representable value
```

11. URN Format

11.1. Specification

SayWhere defines a canonical URN format ([RFC8141] compliant):

urn:saywhere:{language}:{word1}.{word2}.{word3}.{wordN}.{checksum}[:{altitude}]

11.2. Components

* urn:saywhere - Namespace identifier (fixed)

* {language} - ISO 639-1 language code (2 lowercase letters)

* {word1}.{word2}.{word3}.{checksum} - Word phrase

* {altitude} - OPTIONAL signed integer altitude units (unitless)

The terminal checksum is MUST be from a distinct wordlist to the location words to enable the parser to determine if the final word in the word phrase is a location word or the terminal checksum.

11.3. Examples

```
# Without terminal checksum
urn:saywhere:en:grape.column.hip

# With terminal checksum
urn:saywhere:en:grape.column.hip.seal

# With altitude: 20 units (= 60m with 3.0m steps)
urn:saywhere:en:grape.column.hip.seal:20

# Underground: -5 units (= -15m with 3.0m steps)
urn:saywhere:en:grape.column.hip.seal:-5

# Spanish wordlist with checksum
urn:saywhere:es:manzana.montana.atardecer.rojo

# French wordlist with checksum and altitude
urn:saywhere:fr:pomme.montagne.coucher.rouge:15
```

11.4. Parsing Grammar (ABNF)

```
saywhere-urn = "urn:saywhere:" language ":" words [":" altitude]
language     = 2ALPHA
words        = word "." word "." word "." word
word         = 3*15ALPHA
altitude     = ["-"] 1*4DIGIT  ; Unitless integer

ALPHA        = %x61-7A  ; lowercase a-z
DIGIT        = %x30-39  ; 0-9
```

12. Security Considerations

This section follows the guidelines in [RFC3552] for security considerations in protocol specifications.

SayWhere phrases encode precise geographic locations and can reveal sensitive information about individuals' whereabouts. Privacy considerations include:

* *Data Minimization*: Implementations SHOULD NOT log or store phrases without explicit user consent

* *User Awareness*: Users SHOULD be warned that phrases reveal precise physical locations that persist over time

* *Sharing Controls*: Applications SHOULD implement access controls before transmitting or displaying location phrases

* *Hierarchical Disclosure*: Applications can leverage hierarchical phrases to enable progressive disclosure, sharing only the precision level necessary for the use case

* *Pervasive Monitoring*: Per [RFC7258], implementations should consider that location data transmission may be subject to pervasive monitoring attacks

Location phrases shared in public forums, radio communications, or other non-confidential channels should be considered public information accessible to any party.

SayWhere provides two layers of error detection (per-word LSB parity and optional terminal checksum), but neither provides security:

* *No Authentication*: Checksums do NOT provide cryptographic authentication or integrity protection

* *Accidental Errors Only*: Checksums ONLY detect accidental transmission/transcription errors, not deliberate tampering

* *Not Security Critical*: Do not rely on checksums for security-critical applications requiring authenticated location data

* *Malicious Modification*: An attacker can easily generate valid phrases with correct checksums for arbitrary locations

* *Limited Entropy*: Terminal checksum uses only 5 bits (32 values), providing ~3% false negative rate

The terminal checksum is designed for *error detection* (communication integrity), not *security* (adversarial protection).

Applications requiring authenticated location assertions should implement additional cryptographic signatures or message authentication codes.

Implementations depend on wordlist consistency for correct operation:

* *Verification Required*: Implementations MUST verify wordlist integrity using cryptographic checksums (SHA-256 or better recommended)

* *Corruption Impact*: Corrupted or modified wordlists produce incorrect encodings that may place users in unintended or dangerous locations

* *Supply Chain Security*: Consider signing wordlist files for distribution and verifying signatures before use

* *Version Control*: Implementations should verify the wordlist version matches the expected version for the encoding/decoding operation

A compromised wordlist could enable man-in-the-middle attacks where locations are systematically mistranslated.

Implementations should protect against resource exhaustion:

* *Rate Limiting*: Implementations SHOULD rate-limit encoding/decoding operations, particularly in network-facing services

* *Batch Size Limits*: Batch operations SHOULD enforce maximum size limits to prevent memory exhaustion

* *Early Validation*: Invalid input SHOULD be rejected early in processing to avoid expensive computation

12.1. Intellectual Property Considerations

The authors are aware that proprietary word-based geocoding systems exist, including systems covered by patents held by What3Words Limited (e.g., CA2909524A1, GB2540897B, and related patents in multiple jurisdictions). These patents claim specific technical approaches to converting geographic coordinates into word phrases.

SayWhere uses fundamentally different technical mechanisms that distinguish it from these proprietary systems:

12.1.1. Technical Distinctions

1. *Prior Art Foundation*: SayWhere combines two established public domain technologies:
    * Geohash spatial indexing (Niemeyer, 2008) for coordinate encoding
    * BIP-39 mnemonic wordlist (Palatinus et al., 2013) for word mapping
2. *Hierarchical Variable-Length Encoding*: SayWhere uses 1-6+ word phrases with true prefix relationships, where shorter phrases represent spatial areas containing all longer phrases with the same prefix. This hierarchical property is fundamentally incompatible with fixed-length systems.
3. *Direct Geohash Mapping*: SayWhere uses deterministic chunk-based mapping (2 geohash characters → 1 word) without shuffling algorithms, attractiveness ratings, or cell-decomposition transformations.
4. *Error Detection vs. Error Magnification*: SayWhere provides two-layer error detection (per-word parity + optional terminal checksum) rather than error magnification through shuffling.
5. *Open Intermediate Representation*: The geohash intermediate representation is directly observable and reversible, unlike proprietary integer-based cell decomposition schemes.

12.1.2. Mathematical Incompatibility

The hierarchical prefix relationship (where "grape" contains "grape.column" which contains "grape.column.hip") is mathematically incompatible with shuffling-based approaches that intentionally disperse similar inputs to distant outputs. This fundamental design choice ensures SayWhere operates in a different technical space.

12.1.3. Patent Landscape

The authors believe SayWhere's use of published prior art (geohash + BIP-39) combined with hierarchical variable-length encoding represents a distinct technical approach not covered by known third-party patents. However, the authors make no legal determination regarding patent validity, enforceability, or scope.

Implementors are encouraged to:

1. Consult legal counsel regarding patent implications in their jurisdiction
2. Review the technical differences documented in this specification
3. Note that geohash (2008) and BIP-39 (2013) predate many word-based geocoding patents
4. Consider that the hierarchical variable-length approach differs fundamentally from fixed-length cell-decomposition methods

12.1.4. Defensive Publication

This specification serves as a defensive publication documenting the combination of geohash spatial indexing with BIP-39 word encoding for hierarchical variable-length location phrases. The reference implementation is released under GNU GPL v3, ensuring perpetual public availability.

Per RFC 8179, the IETF makes no determination about the validity or applicability of any claimed intellectual property rights. This disclosure is provided for informational purposes to assist implementors in making informed decisions.

13. IANA Considerations

This section follows the guidelines in [RFC8126] for IANA considerations sections.

This document requests the registration of the "saywhere" URN namespace in the "Uniform Resource Names (URN) Namespaces" registry, in accordance with the procedures defined in [RFC8141].

Per [RFC8141] Section 6, the following registration template is provided:

Namespace ID: saywhere

Registration Information: Version 1, 2025-10-12

Declared registrant: SayWhere Contributors

Anuna Research Pty Ltd

Email: contact@saywhere.org

URI: https://codeberg.org/anuna/saywhere

Declaration of syntactic structure: See Section 9 of this document

Relevant ancillary documentation: This document serves as the specification for the saywhere URN namespace

Identifier uniqueness considerations: Uniqueness is guaranteed by the deterministic encoding algorithm specified in this document. Each unique geographic coordinate (latitude, longitude, altitude) and wordlist language combination produces a unique URN.

Identifier persistence considerations: SayWhere URNs remain valid and decodable as long as the corresponding wordlist version is available. Wordlists follow semantic versioning and are maintained in public repositories for long-term availability.

Process of identifier assignment: SayWhere URNs are not centrally assigned. Any party can generate a valid SayWhere URN by applying the deterministic encoding algorithm specified in this document to geographic coordinates.

Process for identifier resolution: SayWhere URNs are decoded to geographic coordinates using the deterministic decoding algorithm specified in Section 7 of this document. Resolution requires access to the appropriate wordlist version.

Rules for Lexical Equivalence: Two SayWhere URNs are lexically equivalent if and only if they are identical after Unicode normalization (NFC) and case normalization (lowercase). The comparison is case-insensitive and performed on Unicode-normalized strings.

Conformance with URN Syntax: SayWhere URNs conform to the URN syntax specified in [RFC8141]. The syntax is formally defined using ABNF in Section 9 of this document.

Validation mechanism: Validation consists of: (1) syntax validation

per the ABNF grammar, (2) verification that each location word exists in the specified language wordlist, (3) validation of per-word LSB parity checksums, and (4) if present, validation of the terminal checksum word as specified in Section 10.

Scope: Global

14. Implementation Guidance

14.1. Performance Optimization

In-Memory Wordlist:

```
# Load wordlist(s) once at startup
wordlist = load_wordlist("saywhere-en-v1.0.0.json")

# Create bidirectional lookup dictionaries
word_to_index = {}
index_to_word = {}

FOR index, word IN wordlist:
    word_to_index[word] = index
    index_to_word[index] = word

# Use for O(1) lookups during encoding/decoding
```

14.2. Testing Requirements

Implementations MUST pass:

1. All test vectors in Appendix B
2. Roundtrip tests (encode → decode → verify coordinates match)
3. Per-word LSB parity validation tests
4. Terminal checksum validation tests (detection and validation)
5. Error handling tests (both parity and checksum errors)
6. Boundary condition tests (poles, dateline, altitude extremes)
7. Altitude quantization tests (different step sizes)
8. Checksum error detection tests (substitution, omission, transposition)

14.3. Reference Implementation

A reference implementation in Scheme (Guile) is available for testing and validation purposes. This implementation is provided as an example and is not required for conformance:

* Repository: https://codeberg.org/anuna/saywhere (https://codeberg.org/anuna/saywhere)

* License: GNU GPL v3

Implementations in other languages that pass the test vectors in Appendix B are considered conformant.

14.4. Web Application Demo

An interactive web application is available for exploring SayWhere encoding and decoding:

* Demo: https://saywhere.org/grape/column/hip (https://saywhere.org/grape/column/hip)

The demo allows users to:

* Visualize hierarchical phrase resolution on an interactive map

* Encode coordinates to word phrases

* Decode word phrases to geographic locations

* Explore the spatial containment relationships between different precision levels

* Test error detection capabilities

15. References

15.1. Normative References

[RFC8141] Saint-Andre, P. and J. Klensin, "Uniform Resource Names (URNs)", RFC 8141, DOI 10.17487/RFC8141, April 2017, <https://www.rfc-editor.org/rfc/rfc8141>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

15.2. Informative References

[RFC3552] Rescorla, E. and B. Korver, "Guidelines for Writing RFC Text on Security Considerations", RFC 3552, BCP 72, July 2003, <https://www.rfc-editor.org/info/rfc3552>.

[RFC7258] Farrell, S. and H. Tschofenig, "Pervasive Monitoring Is an Attack", RFC 7258, BCP 188, May 2014, <https://www.rfc-editor.org/info/rfc7258>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", RFC 8126, BCP 26, June 2017, <https://www.rfc-editor.org/info/rfc8126>.

[BIP39] Palatinus, M., Rusnak, P., Voisine, A., and S. Bowe, "BIP 0039: Mnemonic code for generating deterministic keys", September 2013, <https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki>.

[GEOHASH] Niemeyer, G., "Geohash", 2008, <https://en.wikipedia.org/wiki/Geohash>.

[GEOWORDS] mfrank, "GeoWords", n.d., <https://mfrank.codeberg.page/GeoWords>.

Appendix A. Geohash Algorithm Details

A.1. Encoding Pseudocode

```
   FUNCTION geohash_encode(latitude, longitude, precision):
       lat_min = -90.0
       lat_max = 90.0
       lon_min = -180.0
       lon_max = 180.0

       bits = []
       is_longitude = true

       # Generate precision * 5 bits (base32 = 5 bits per char)
       WHILE length(bits) < (precision * 5):
           IF is_longitude:
               mid = (lon_min + lon_max) / 2
               IF longitude > mid:
                   bits.append(1)
                   lon_min = mid
               ELSE:
                   bits.append(0)
                   lon_max = mid
           ELSE:
               mid = (lat_min + lat_max) / 2
               IF latitude > mid:
                   bits.append(1)
                   lat_min = mid
               ELSE:
                   bits.append(0)
                   lat_max = mid

           is_longitude = NOT is_longitude

       # Convert bits to base32 string
       base32_chars = "0123456789bcdefghjkmnpqrstuvwxyz"
       geohash = ""

       FOR i = 0 TO length(bits) STEP 5:
           chunk = bits[i:i+5]
           value = binary_to_decimal(chunk)
           geohash += base32_chars[value]

       RETURN geohash
   END FUNCTION
```

Appendix B. Test Vectors

B.1. Hierarchical Encoding Test Cases

B.1.1. Test Case 1: New York City

```
 Input:
   latitude: 40.7128
   longitude: -74.0060
   max_precision: 3
   wordlist_version: "1.0.0"

 Expected Output (Hierarchical):
   geohash: "dr5reg"
   phrases: [
     "grape",
     "grape.column",
     "grape.column.hip"
   ]

 Verification:
   - Each phrase is a prefix of the next phrase
   - Decoding "grape.column" returns coordinates within NYC metro area
   - Decoding "grape.column.hip" returns precise location
```

B.1.2. Test Case 2: London

```
   Input:
     latitude: 51.5074
     longitude: -0.1278
     max_precision: 4
     wordlist_version: "1.0.0"

   Expected Output:
     geohash: "gcpvj0du"
     phrases: [
       "kit",
       "kit.puzzle",
       "kit.puzzle.marine",
       "kit.puzzle.marine.grit"
     ]

   Prefix Tests:
     - "kit" contains all locations starting with "kit.*"
     - "kit.puzzle" ⊂ "kit"
     - "kit.puzzle.marine" ⊂ "kit.puzzle"
```

B.1.3. Test Case 3: Equator Crossing

```
   Input:
     latitude: 0.0
     longitude: 0.0
     max_precision: 3
     wordlist_version: "1.0.0"

   Expected Output:
     geohash: "7zzzzz"
     phrases: [
       "divert",
       "divert.zone",
       "divert.zone.zone"
     ]

   Note: Tests edge case at equator and prime meridian
```

B.1.4. Test Case 4: With Terminal Checksum

```
  Input:
    latitude: 40.7128
    longitude: -74.0060
    num_words: 3
    include_checksum: true

  Expected Output:
    location_phrase: "grape.column.hip"

    Checksum calculation:
      Word indices: [814, 366, 863]
      Packed bytes: [0xCC, 0xDB, 0x6F]
      CRC-8 (polynomial 0x07, init 0xFF): 0x1E
      Checksum index: 0x1E % 32 = 30
      Checksum: "seal"

    Full phrase: "grape.column.hip.seal"

  Validation:
    - Decoding detects "seal" is from checksum vocabulary
    - Computes checksum for ["grape", "column", "hip"] → "seal"
    - Checksum validates, returns coordinates with checksum_valid=true

  Error Detection Test:
    Corrupted: "grape.color.hip.seal" (column→color)
    Expected checksum for ["grape", "color", "hip"] → "whale" ≠ "seal"
    Validation fails, error detected
```

Acknowledgments

The SayWhere system uses the geohash algorithm [GEOHASH] created by Gustavo Niemeyer and the BIP-39 wordlist [BIP39] developed by the Bitcoin community. The altitude component was inspired by the GeoWords [GEOWORDS] implementation by mfrank. We thank these contributors for their foundational work.

Special thanks to the open-source community for feedback and testing of the SayWhere implementation.

Author’s Address

Hugo O’Connor
Anuna Research Pty Ltd
Australia
Email: hugo@anuna.io
URI: https://codeberg.org/anuna/saywhere