$\mathbb{A}$bstract $\mathbb{L}$ogical $\mathbb{A}$ssembly $\mathbb{N}$etwork (**ALAN**)

A Transition-Rule-Independent High-Performant Trustless State Machine

(Named after Dr.Alan Mathison Turing)

JOBY REUBEN
Auguth Research Foundation
Bangalore, India
joby@auguth.org

ABSTRACT

We present the formal specification of Alan

PREFACE

Preface from Joby Reuben

FOREWORD

AUTHOR
Designation, Affiliated Organization

Contents of Foreword here

Contents

I. INTRODUCTION

A. Goals
B. Outcomes
C. Architecture
D. Terminologies
E. Overview of Handbook

II. PRELIMINARIES

A. Data Representation

1) Byte Array

Byte Array or *Sequence of Bytes*, denoted as $\mathbb{B}$ of length n is referred as

$$\mathbb{B}_n = \{\mathbb{B}_0, \mathbb{B}_1, \ldots, \mathbb{B}_{n-1}\}$$

$\mathbb{B}$ is in the form - of conventional zero-based indexing where the first element of the set is represented as the zeroth element. Additionally, $\mathbb{B}_i$, a single byte shall contain 8-bits $(b_i \in \mathbb{B}_i)$ where, each bit $b_i \in \{0, 1\}$. In decimal representation i.e., `base-10` each byte shall be $0 \leq \mathbb{B}_i \leq 255$.

The concatenation of byte arrays $A = \{A_0, A_1, \ldots, A_{n-1}\}$ of length n, and $B = \{B_0, B_1, \ldots, B_{m-1}\}$ of length m is referred as

$$A \parallel B := \{A_0, A_1, \ldots, A_{n-1}, B_0, B_1, \ldots, B_{m-1}\}$$

Throughout the inner sections of the document, a byte array is described as a vector type (see Section II-B3.H) of an 8-bit unsigned integer `u8` (see Section II-B3.B) elements i.e., `vec[u8]` or an array of `u8` elements i.e., `ary[u8:N]` encoded in little endian III-A1 format.

2) Bitwise Representation

For a given byte $0 \leq \mathbb{B} \leq 255$ in decimal representation i.e., `base-10`, the byte's *bitwise representation* in bits $b_i \in \{0, 1\}$ i.e., `base-2` is defined as:

$$\mathbb{B} = \{b_7, \ldots b_0\}$$

Alternatively we can denote a bit of a byte using

$$b^{\text{byte index}}_{\text{bit index}}$$

This representation implies that the bits b_7 through b_0 constitute the binary representation of the byte $\mathbb{B}$. In binary representation, the bits are typically indexed from right to left, starting with the *least significant bit* (LSb) at position 0, and increasing toward the *most significant bit* (MSb).

Most Significant Bits carry higher values such as $2^7 = 128$ compared to Least significant Bits $2^4 = 16$. Therefore, b_7 - one of 8 bits of a byte represents the leftmost bit (MSb), and b_0 represents the rightmost bit (LSb).

Conversion of bitwise representation to decimal notation shall be defined as

$$\mathbb{B} = \{2^7 \cdot b_7, \ldots, 2^0 \cdot b_0\}$$

3) Hexadecimal Representation

4-bits or a *nibble* can be represented in hexadecimal or `base-16` format for readability as opposed to `base-2` binary format, with a prefix of 0x || HEX-DIGITS to denote `base-16`.

Each byte will contain 2 hexadecimal values, where,

$$\mathbb{B} = \textsc{Hex-Encode}(\{b_7, \ldots, b_4\}) \parallel \textsc{Hex-Encode}(\{b_3, \ldots, b_0\})$$

Binary	0000	0001	0010	0011	0100	0101	0110	0111
Hex	0	1	2	3	4	5	6	7

Binary	1000	1001	1010	1011	1100	1101	1110	1111
Hex	8	9	A	B	C	D	E	F

4) UTF-8

UTF is an 8-bit byte array II-A1 per character encoding system, extending the ASCII text which is only able represents a single byte. UTF-8 can represent more international languages, and Unicode characters/symbols. For non-ASCII characters such as symbols, emojis, etc, UTF-8 characters will have multiple bytes to represent the character.

A Character's unicode can be fetched here for reference, and more detailed description of the standard is given in its wiki-page. As UTF encoding is a standard followed by most internet protocols, strings, and characters are interpreted, and stored as UTF bytes.

B. Logic Constructs

Throughout ALAN's specifications, functions, math expressions, and algorithms are written using the logic constructs defined here. These constructs form a foundational framework for creating near-high-level pseudocode, and pseudo-functions, which directly assists in evaluating its suitability for implementation. Implementations are encouraged to follow to the algorithms, and structures while developing language-specific executable code, ensuring ease of maintainability. As this specification handbook is language-agnostic, and incorporates higher-level logic constructs, it serves as a comprehensive reference for ALAN's diverse implementations.

1) Operators

Operators are symbolic representations used to denote specific operations or relationships between two or more elements in logics.

- $\wedge$: Denotes `AND` Operator
- $\vee$: Denotes `OR` Operator

2) Set Notations

Set notations are fundamentals for defining, and manipulating collections of objects. These notations provide a precise way to describe relationships between elements, and operations within sets. Each symbol carries specific meanings, and rules that aid in expressing set-related concepts for effectively implementing the Alan's Specification.

- $a(A(\mathbb{A}))$: a belongs to set A, and A belongs to set $\mathbb{A}$
- $\in$: Represents *element of* E.g., $i \in I$ denotes i is an element of set I
- $i \in b_i(\mathbb{B}) : i > 0$ Set builder notation which constructs a set, here the set includes all elements b_i with condition $i > 0$ of set $\mathbb{B}$. Here the partition : denotes *such that*
- $\parallel$: Denotes concatenation operation of two sets or an element to a set.
- $\emptyset$: Represents an empty set with an empty value or absence of data
- $\mid S \mid$: Represents the cardinality of the set i.e., the number of elements in the set S
- $S \leftarrow S \setminus e_i$: Represents the set minus operation i.e., $\setminus$ where the element e_i is removed from the set S. This operation does not imply assigning a zero value, as all elements indexed by $e_0, \ldots, e_n$ in set S are updated to reflect the removal of e_i.

3) Types

For each individual piece of data required for executing software, or resulting from its operations, attributes such as size, and the range of possible values are crucial. Types ensure data safety, integrity, and facilitate efficient memory usage. A `type` specifies how a hardware interprets, and manipulates data in memory, providing essential information for memory allocation, data representation, and ensuring correct operation according to predefined rules, and constraints. The specification defines various types to enhance logic, and define their attributes.

II-B3.A. Bool:

A Boolean value denoted by `bool` returns either `1 ∨ 0` i.e., `true ∨ false`

$$\mathtt{bool} \in \{0, 1\}$$

II-B3.B. Unsigned Integer:

An Unsigned Integer, denoted by `uN`, where `N` can take values such as $\{8, 16, 32, 64, 128\}$ represents a non-negative integer where its byte length is based on its total bits `N`. It is normally represented in big endian format $u = \mathbb{B}_0, \ldots, \mathbb{B}_n$.

- Unsigned Integer 8 bit : $\mathtt{u8} = 0 \leq \mathtt{u8} < 2^8$
- Unsigned Integer 16 bit : $\mathtt{u16} = 0 \leq \mathtt{u16} < 2^{16}$
- Unsigned Integer 32 bit : $\mathtt{u32} = 0 \leq \mathtt{u32} < 2^{32}$
- Unsigned Integer 64 bit : $\mathtt{u64} = 0 \leq \mathtt{u64} < 2^{64}$
- Unsigned Integer 128 bit : $\mathtt{u128} = 0 \leq \mathtt{u128} < 2^{128}$
- Unsigned Integer size : `size` depends on target architecture i.e., 64 bit - `u64` or 32 bit - `u32`

II-B3.C. Signed Integer:

A Signed Integer, denoted by `iN`, where `N` can take values such as $\{8, 16, 32, 64, 128\}$, represents integers that can be positive, negative, or zero. Its byte length is based on its total bits `N`. In case of negative integer, it is typically represented in two's complement format.

- Signed Integer 8-bit: $\mathtt{i8} = -2^7 \leq \mathtt{i8} < 2^7$
- Signed Integer 16-bit: $\mathtt{i16} = -2^{15} \leq \mathtt{i16} < 2^{15}$
- Signed Integer 32-bit: $\mathtt{i32} = -2^{31} \leq \mathtt{i32} < 2^{31}$
- Signed Integer 64-bit: $\mathtt{i64} = -2^{63} \leq \mathtt{i64} < 2^{63}$
- Signed Integer 128-bit: $\mathtt{i128} = -2^{127} \leq \mathtt{i128} < 2^{127}$
- Signed Integer size : `size` depends the target architecture i.e., 64 bit - `u64` or 32 bit - `u32`

II-B3.D. Floating Integer:

A Float, denoted by `fN`, where `N` can take values such as $\{32, 64\}$, represents real numbers $\mathbb{R}$ using floating-point representation. Floating-point numbers are used to represent numbers that may have a fractional part or require a larger range than integers.

- Float 32-bit (Single Precision): $\mathtt{f32} = \leq \mathtt{f32} <$
- Single-precision floating-point numbers (32-bit) provide approximately 7 decimal digits or 23 binary digits of precision.
- Float 64-bit (Double Precision): $\mathtt{f64} = \leq \mathtt{f64} <$
- Double-precision floating-point numbers (64-bit) provide approximately 15-16 decimal digits or 53 binary digits of precision.

II-B3.E. Character:

A `char` type typically represents a single Unicode character encoded in UTF-8 encoded format II-A4. In UTF-8 encoding, characters are typically 4 bytes in length with a range to represent 1,048,576 unicode characters.

II-B3.F. String:

A String i.e., `str` type represents a sequence of unicode characters with its empty bytes excluded (assumption).

II-B3.G. Array:

An Array is a fixed collection of values of same type `ary[type:N]`, where `N` denotes the number of elements of same type.

II-B3.H. Vector:

A Vector is a dynamic array of same type `vec[type]` where the length is not fixed.

II-B3.I. Tuple:

A Tuple is a fixed collection of values of different types `tup`

4) Function

A function represented with a `Function Name` encapsulates a set of instructions designed to perform a specific task or computation. A function may or may not accept inputs, processes them according to its defined logic, and returns a specific output. For instance, a function might calculate the sum of two integers a, and b, where both a, and b's types are defined along with its output value's type.

Inputs of a function, also known as parameters, are variables provided to the function when it is called or executed. These inputs specify the data that the function will operate on. For example, provided that there is a sum function $\textsc{Sum}(a, b)$, where a, and b are inputs of type `u8` i.e., $\textsc{Sum}(a^{\mathtt{u8}}, b^{\mathtt{u8}})$.

The return value of a function specifies the data type of the result it produces after execution. In the sum function, the return type `u8` i.e., $\mathrm{SUM}(a^{\mathtt{u8}}, b^{\mathtt{u8}}) \rightarrow k^{\mathtt{u8}}$ indicates that the function will return an unsigned integer value with bounds of 8-bits.

5) Conditional Statements

Conditional statements within a function, control the flow of execution based on given conditions. These statements include `if (condition) (logic)`, `else if (condition) (logic)` (optional), and `else (logic)` which allow functions to evaluate different conditions, and execute specific blocks of logic that require different actions based on varying scenarios.

6) Loop Constructs

Loop constructs allow for repetitive execution of a block of instructions within a function until a specific condition is met. The loop terminates if the condition is not met. Here are different types of loops:

1) `for all (condition) (logic)`: Iterating over all elements of a set or collection when the condition or scenario defined is met.
2) `while (condition) do (logic)`: Logic execution based on conditions provided is met.

7) Storage

Variables in algorithms, and functions prefixed with tilde : $\tilde{v}$ denote memory variables that undergo garbage collection. This implies that such variables are erased from memory after the function where $\tilde{v}$ is utilized terminates i.e., when the variable goes out of its memory scope. In contrast, variables lacking the $\tilde{}$ symbol are intended to be stored persistently in storage memory. These persistent variables can be retrieved by the function whenever they are needed for subsequent operations.

8) Concurrency

The algorithms described in this specification include a logic construct designed to define concurrent instructions, which can enhance computational performance by dividing instructions among multiple threads. This construct is denoted by `concurrent(n)`, where `n` specifies the number of concurrent threads required. Following this, a set of individual instructions or operations is provided that can independently execute on separate threads. The concurrent process concludes once all operations across these threads have completed. Algorithms can be evaluated for grouping of instructions under a concurrent operation based on write access to distinct memory or storage locations.

III. SERIALIZATION

A. Encoding

Encoding is used to represent data in a serialized format, which ensures data integrity, security, and efficiency when sending over networking protocols or storing in databases for later retrieval.

The Encoding Specification described in this document is a modified-derivative of SCALE-Encoding (Simple Concatenated Aggregate LittleEndian) by Parity Technologies used as the defacto encoding for Substrate-Framework to encode byte arrays and other data structures. SCALE provides a canonical encoding to produce consistent hash values across their implementation, including the State Database (see Section IV-A).

1) Little Endian

Little-endian is a byte order where the least significant byte (LSB) is stored at the lowest memory address, and subsequent bytes are stored at higher memory addresses.

A non-negative integer $\mathbb{Z}$ is expressed as a sequence of bytes (see Section II-A1), where each byte $\mathbb{B}_i$ satisfies $0 \leq \mathbb{B}_i \leq 255$ i.e., `base-256`. In *little-endian* format, $\mathbb{Z}$ is represented as:

$$\mathbb{Z} = \{\mathbb{B}_0, \dots, \mathbb{B}_n\}$$

This can be interpreted as:

$$\{\mathbb{B}_n, \dots, \mathbb{B}_0\} \leftarrow \textsc{LittleEndian}(\{\mathbb{B}_0, \dots, \mathbb{B}_n\})$$

Examples of Little-Endian are provided in Section A1.

2) Length Encoding

Length encoding of a non-negative number $n \in \mathbb{N} \rightarrow \mathbb{B}$ represented in a byte array is denoted as LENGTHENCODE. Length encoding is used to encode integer numbers of varying sizes. The encoding process of the non-negative number n is divided into different cases based on its magnitude:

- Case 1 (1 Byte): $0 \leq n < 2^6$ Representation: $b_1^0, b_0^0 = 0, 0$
- Case 2 (2 Bytes): $2^6 \leq n < 2^{14}$ Representation: $b_1^0, b_0^0 = 0, 1$
- Case 3 (4 Bytes): $2^{14} \leq n < 2^{30}$ Representation: $b_1^0, b_0^0 = 1, 0$
- Case 4 ($m + 1$ Bytes): $n \geq 2^{30}$ Representation: $b_1^0, b_0^0 = 1, 1$, and $b_7^0, \dots, b_2^0 = m$, where m denotes the length of the total bytes of the original non-negative integer before encoding.

Examples of Length Encoding are provided in Section A2.

III-A2.A. LengthEncode():

- Algorithm 1 encodes a non-negative integer (unsigned integer `u128,u64,u32`) into a byte array format (`vec[u8]`).
- Implementations should include additional necessary variants for the remaining unsigned integer data types i.e., `u16,u8`
  - LENGTHENCODE($n^{\texttt{u16}}$) , where the case $n \geq 2^{30}$ can be omitted
  - LENGTHENCODE($n^{\texttt{u8}}$) , where the cases $n \geq 2^{30}$, $2^{14} \leq n < 2^{30}$ can be omitted
- Test cases for Algorithm 1 are provided in Section IX-A1.

3) Varying Data Type

A Varying Data Type $\mathcal{T}$ is an ordered set of data types denoted by,

$$\mathcal{T} = \{T_0, \dots, T_n\}$$

A value A is represented as A_{val}, belonging to a pair $\{A_{\text{type}}, A_{\text{val}}\}$, where $A_{\text{type}} = i \in T_i(\mathcal{T})$, indicating that A_{val} is of the individual data type T_i within the ordered set of all data types $\mathcal{T}$.

The value A_{val} of a certain type T_i can have a zero value, but should not be empty, as for sequences with zero elements - its length ie., number of elements shall describe its empty value (see Section III-A10.A).

Algorithm 1 LengthEncode() $\mathcal{O}()$

function LENGTHENCODE($n^{\texttt{u128} \vee \texttt{u64} \vee \texttt{u32}}$)
  $\tilde{n}^{\texttt{vec[u8]}} \leftarrow$ LEBYTEARRAY(n) ▷ VIII-B1
  if $2^0 \leq n < 2^6$ **then**
    $\tilde{p}^{\texttt{vec[u8]}} \leftarrow$ SHIFTLEBYTEARRAYBITTOMSB ($\tilde{n}, 2$) ▷ VIII-C1
    concurrent 2
      $b_1(\texttt{u8}_0(\tilde{p})) \leftarrow$ `false` ▷ II-A2
      $b_0(\texttt{u8}_0(\tilde{p})) \leftarrow$ `false` ▷ II-A2
    end concurrency
  else if $2^6 \leq n < 2^{14}$ **then**
    $\tilde{p}^{\texttt{vec[u8]}} \leftarrow$ SHIFTLEBYTEARRAYBITTOMSB ($\tilde{n}, 2$) ▷ VIII-C1
    concurrent 2
      $b_1(\texttt{u8}_0(\tilde{p})) \leftarrow$ `false` ▷ II-A2
      $b_0(\texttt{u8}_0(\tilde{p})) \leftarrow$ `true` ▷ II-A2
    end concurrency
  else if $2^{14} \leq n < 2^{30}$ **then**
    $\tilde{p}^{\texttt{vec[u8]}} \leftarrow$ SHIFTLEBYTEARRAYBITTOMSB ($\tilde{n}, 2$) ▷ VIII-C1
    concurrent 2
      $b_1(\texttt{u8}_0(\tilde{p})) \leftarrow$ `true` ▷ II-A2
      $b_0(\texttt{u8}_0(\tilde{p})) \leftarrow$ `false` ▷ II-A2
    end concurrency
  else if $n \geq 2^{30}$ **then**
    $\tilde{q}^{\texttt{u-size}} \leftarrow$ LENBYTES(n) ▷ VIII-A1
    $\tilde{k}^{\texttt{vec[u8]}} \leftarrow$ LEBYTEARRAY($\tilde{q}$) ▷ VIII-B1
    $\tilde{y}^{\texttt{vec[u8]}} \leftarrow$ SHIFTLEBYTEARRAYBITTOMSB ($\tilde{k}, 2$) ▷ VIII-C1
    concurrent 3
      $b_1(\texttt{u8}_0(\tilde{y})) \leftarrow$ `true` ▷ II-A2
      $b_0(\texttt{u8}_0(\tilde{y})) \leftarrow$ `true` ▷ II-A2
      $\tilde{y}^{\texttt{vec[u8]}} \leftarrow \tilde{y}^{\texttt{vec[u8]}} \setminus \texttt{u8}_1(\tilde{y})$ ▷ II-B2
    end concurrency
    $\tilde{p}^{\texttt{vec[u8]}} \leftarrow$ CONCATBYTEARRAYS ($\tilde{y}, \tilde{n}$) ▷ VIII-C2
  end if
   return $\tilde{p}$
end function

The Encoding for a value $A = \{A_{\text{type}}, A_{\text{val}}\}$ of an ordered set of varying data types is defined as,

$$\textsc{Encode}(A) = \textsc{LengthEncode}(i \in T_i(\mathcal{T})) \parallel \textsc{Encode}(A_{\text{val}})$$

Examples of Varying Data Type are given in Section A3

III-A3.A. Index of Data Types:

List of Varying Data Type Indexes (see Figure ??) including special data types (see Section III-A13) according to which FINDDATATYPEINDEX Function (see Algorithm 2) is formulated.

III-A3.B. FindDataTypeIndex():

- Algorithm 2 takes any data type as input, fetches the index of a given varying data type (see Section III-A3), and returns a variable that contains the index n of its data type as a unsigned 8-bit integer i.e., `u8`.
- For each data type the function's variants must be implemented individually and the index must be hardcoded.

Index of Data Types

$$\mathcal{T} = \begin{cases} T_0 & \texttt{none}\ \text{III-A4.}A \\ T_1 & \texttt{some}\ \text{III-A4.}B \\ T_2 & \texttt{ok}\ \text{III-A5} \\ T_3 & \texttt{err}\ \text{III-A5} \\ T_4 & \texttt{bool}\ \text{III-A7 Boolean Type} \\ T_5 & \texttt{char}\ \text{II-B3.}E\ \text{Character Type} \\ T_6 & \texttt{str}\ \text{II-B3.}F\ \text{String Type} \\ T_7 & \texttt{vec}\ \text{II-B3.}H\ \text{Sequence with varying length} \\ T_8 & \texttt{ary}\ \text{II-B3.}G\ \text{Sequence with fixed length} \\ T_9 & \texttt{tup}\ \text{II-B3.}I\ \text{Tuple} \\ T_{10} & \texttt{u8}\ \text{II-B3.}B\ \text{Unsigned-Integer Type} \\ T_{11} & \texttt{u16}\ \text{II-B3.}B\ \text{Unsigned-Integer Type} \\ T_{12} & \texttt{u32}\ \text{II-B3.}B\ \text{Unsigned-Integer Type} \\ T_{13} & \texttt{u64}\ \text{II-B3.}B\ \text{Unsigned-Integer Type} \\ T_{14} & \texttt{u128}\ \text{II-B3.}B\ \text{Unsigned-Integer Type} \\ T_{15} & \texttt{i8}\ \text{II-B3.}C\ \text{Signed-Integer Type} \\ T_{16} & \texttt{i16}\ \text{II-B3.}C\ \text{Signed-Integer Type} \\ T_{17} & \texttt{i32}\ \text{II-B3.}C\ \text{Signed-Integer Type} \\ T_{18} & \texttt{i64}\ \text{II-B3.}C\ \text{Signed-Integer Type} \\ T_{19} & \texttt{i128}\ \text{II-B3.}C\ \text{Signed-Integer Type} \\ T_{20} & \texttt{f32}\ \text{II-B3.}D\ \text{FloatingPoint-Integer Type} \\ T_{21} & \texttt{f64}\ \text{II-B3.}D\ \text{FloatingPoint-Integer Type} \\ T_{22} & \texttt{nib}^{\texttt{u8}}\ \text{III-A13.}A\ \text{Nibble Type (SDT)} \\ T_{23} & \texttt{vec[nib]}\ \text{III-A13.}A\ \text{Nibble Vector Sequence Type (SDT)} \\ T_{24} & \texttt{ary[nib]}\ \text{III-A13.}A\ \text{Nibble Array Sequence Type (SDT)} \end{cases}$$

- The return type is kept `u8` since the current highest index of the index of data types (see Section III-A3.A) is $n < 256$. In case if the index number overflows $n \geq 256$, the return variable's size should be attended to and updated to suitable length.
- Test cases for Algorithm 2 are provided in Section IX-A2.

Algorithm 2 FindDataTypeIndex() $\mathcal{O}()$

function FINDDATATYPEINDEX($n^{\texttt{type}}$)
  return $\tilde{k}^{\texttt{u8}} \leftarrow$ Index from III-A3.A
end function

4) Option Type

The Option type is a varying data type with indices 0, and 1, i.e., $\{T_0, T_1\}(\mathcal{T})$ (see Section III-A3.A). Any individual data type can be an Option type which is typically used when denoting that the data type's value can be `present` $\equiv T_1(\mathcal{T})$ or `absent` $\equiv T_0(\mathcal{T})$. It has two states:

1) `None` $\equiv$ `absent`: indicating the absence of a value of the data type
2) `Some` $\equiv$ `present`: indicating the presence of a value of the data type

III-A4.A. None Type:

For `None` type, the encoding scheme is defined as,

$$\text{ENCODE}(\texttt{None}, \texttt{value}) = 0_{\mathbb{B}_1} = 00000000$$

where, the none type shall be the first index of the Varying Data Type `None` $= T_0(\mathcal{T})$ (see Section III-A3.A). Hence,

$$\text{ENCODE}(A) = \{\text{LENGTHENCODE}(0 \in T_0(\mathcal{T})) \parallel \emptyset\}$$

where, $\emptyset$ is defined as empty set (see Section II-B2)

III-A4.B. Some Type:

For `Some` type, the encoding scheme as is defined as,

$$\text{ENCODE}(\texttt{Some}, \texttt{value})$$

where, the `some` type shall be the index $i = 1$ such that, `Some` $= T_1(\mathcal{T})$ of the Varying Data Type (see Section III-A3.A).

$$\text{ENCODE}(A) = \{\text{LENGTHENCODE}(1 \in T_1(\mathcal{T})) \parallel \text{ENCODE}(A_{\texttt{type}}, A_{\texttt{val}})\}$$

Examples of Some Type are provided in Section A4

5) Result Type

The Result type is a varying data type with indices 2, and 3, i.e., $\{T_2, T_3\}(\mathcal{T})$ (see Section III-A3.A), often used to represent the outcome of an operation or function that can either succeed (`Ok` $\equiv T_2(\mathcal{T})$) or fail (`Err` $\equiv T_3(\mathcal{T})$). It has two states:

1) `1` $\equiv$ `Ok`: indicating success
2) `0` $\equiv$ `Err`: indicating an error or failure

Both types can either contain additional data or be defined as empty types.

Hence, for a function or operation to indicate success (`1`) or failure (`0`):

$$f(x) = \begin{cases} \text{ENCODE}(2 \in T_2(\mathcal{T}), A_{\text{type}}, A_{\text{value}}) & \text{if } \texttt{Ok} \\ \text{ENCODE}(3 \in T_3(\mathcal{T}), A_{\text{type}}, A_{\text{value}}) & \text{if } \texttt{Err} \end{cases}$$

where,

$$\text{ENCODE}(\texttt{Result}, A_{\text{type}}, A_{\text{value}}) = \text{LENGTHENCODE}(3 \geq i \geq 2 \in T_i(\mathcal{T})) \parallel \text{ENCODE}(A_{\text{type}}, A_{\text{value}})$$

Examples of Result Type are provided in Section A5.

6) Empty Type

The encoding scheme for an empty type is defined as a byte array of zero length (empty byte array), depicted as $\emptyset$, where $\emptyset$ represents an empty value or absence of data (see Section II-B2)

$$\text{ENCODE}(\emptyset) = \text{empty byte array}$$

7) Boolean Encoding

For a boolean value b defined as,

$$b \rightarrow \begin{cases} 0 & \text{if } b = \texttt{false} \\ 1 & \text{if } b = \texttt{true} \end{cases}$$

A Boolean value is encoded as a byte $\mathbb{B}$ defined as,

$$\text{BOOLENCODE}(n) = \text{LENGTHENCODE}(4 \in T_4(\mathcal{T})) \parallel \{\{00000000 \text{ if } \texttt{bool} = \texttt{false}\} \vee \{00000001 \text{ if } \texttt{bool} = \texttt{true}\}\}$$

Hence, the boolean value `False` is encoded as a single byte (8 bits) with value 00000000_0, and the boolean value `True` is encoded as a byte with value 1, represented by 00000001_0.

III-A7.A. BoolEncode:

- Algorithm 3 encodes a bool value, and returns a blob of encoded byte array i.e., `vec[u8]`.
- Test cases for Algorithm 3 are provided in Section IX-A3.

Algorithm 3 BoolEncode() $\mathcal{O}()$

function BOOLENCODE($n^{\texttt{bool}}$)
  if $n = \texttt{true}$ **then**
    $\texttt{u8}_0(\tilde{v}^{\texttt{vec[u8]}}) \leftarrow 1$
  else
    $\texttt{u8}_0(\tilde{v}^{\texttt{vec[u8]}}) \leftarrow 0$
  end if
  return $\tilde{v}$
end function

8) Character Encoding

A character is a UTF-8 encoded value, defined as:

$$\texttt{char}(k) = \text{UTF-8}(k)$$

Since a character is encoded to byte array format, and each character's length is self-explanatory, it can be treated as fixed-length integers (see Section III-A9). Therefore, it is encoded in little-endian format (see Section III-A1).

The Character type is a varying data type with index 5, i.e., $T_5(\mathcal{T})$ (see Section III-A3.A):

$$\text{ENCODE}(\texttt{char}) = \text{LENGTHENCODE}(5 \in T_5(\mathcal{T})) \\ \| \text{LITTLEENDIAN}(\text{UTF-8}(\texttt{char}))$$

Examples of Character Type are provided in Section A6.

9) Fixed Length Encoding

The encoding scheme for fixed length integers, and types such as $\{\texttt{u8, u16, u32, u64, u128}\}$, and $\{\texttt{i8, i16, i32, i64, i128, f32, f64, char}\}$ is straightforward. These types have a fixed size where the decoder knows the length, and the encoding scheme is equivalent to little-endian encoding (see Section III-A1) of those values.

10) Sequence Encoding

A *Sequence* S is defined as a collection of elements A_i of the same type, where sequences denote vectors (see Section II-B3.H), and arrays (see Section II-B3.G):

$$S = \{A_0, \ldots, A_i, \ldots, A_n\}$$

The encoding process of a sequence involves encoding the length of the sequence, denoted as LENGTHENCODE($|S|$), which is the number of elements in the set. This is followed by the variable data type encoding of the first element of the sequence, and then the value encoding (encoding without data type index) of individual elements VALUEENCODE(A_i) (see Section III-A14.B).

The Sequence type is a varying data type with indices 7, and 8, i.e., $\{T_7, T_8\}(\mathcal{T})$ (see Section III-A3.A):

$$\text{SEQUENCEENCODE}(S) = \text{LENGTHENCODE}(8 \geq (i \in T_i(\mathcal{T})) \geq 7) \\ \| \text{LENGTHENCODE}(|S|) \| \\ \text{LENGTHENCODE}(i \in T_i(\mathcal{T})) \| \text{VALUEENCODE}(A_0) \ldots \\ \ldots \| \text{VALUEENCODE}(A_n)$$

A special case for Sequences that includes either of Option Type (see Section III-A4) or Result Type (see Section III-A5) data types of index $3 \geq i \geq 0$ (see Section III-A3.A) is defined. Instead of VALUEENCODE(A_i), the elements are individually encoded:

$$\text{SEQUENCEENCODE}(S) = \text{LENGTHENCODE}(8 \geq (i \in T_i(\mathcal{T})) \geq 7) \\ \| \text{LENGTHENCODE}(|S|) \| \\ \text{ENCODE}(A_0) \ldots \| \text{ENCODE}(A_n)$$

Examples of Sequence Type are provided in Section A7.

III-A10.A. SequenceEncode():

- Algorithm 4 encodes a sequence i.e., data types `ary` (see Section II-B3.G) or `vec` (see Section II-B3.H), and returns a blob of encoded byte array i.e., `vec[u8]`.
- The return type of function call inside the algorithm : FINDDATATYPEINDEX (see Section III-A3.B) is kept `u8` since the current highest index of the index of data types (see Section III-A3.A) is $n < 256$. In case if the index number overflows $n \geq 256$, the return variable's size should be attended to and updated to suitable length.
- Test cases for Algorithm 4 are provided in Section IX-A4.

Algorithm 4 SequenceEncode() $\mathcal{O}()$

function SEQUENCEENCODE($S^{\texttt{ary} \vee \texttt{vec}}$)
  $\tilde{i}^{\texttt{u-size}} \leftarrow$ NUMELEMENTS(S) ▷ VIII-A2
  if $\tilde{i} > 0$ **then**
    $\tilde{x}^{\texttt{u8}} \leftarrow$ FINDDATATYPEINDEX 2($e_0(S)$) ▷ III-A3.B
    if $3 \geq \tilde{x} \geq 0$ **then**
      $\tilde{n}^{\texttt{vec[u8]}} \leftarrow$ITERATEENCODEFORSEQUENCE 5(S) ▷ III-A10.B
    else
      concurrent 2
        $\tilde{y}^{\texttt{vec[u8]}} \leftarrow$ LEBYTEARRAY($\tilde{x}$) ▷ VIII-B1
        $\tilde{a}^{\texttt{vec[u8]}} \leftarrow$ITERATEVALUEENCODEFORSEQUENCE 6($S, \tilde{x}$) ▷ III-A10.C
      end concurrency
      $\tilde{n}^{\texttt{vec[u8]}} \leftarrow$ CONCATBYTEARRAYS ($\tilde{y}, \tilde{a}$) ▷ VIII-C2
    end if
  else
    $\tilde{n}^{\texttt{vec[u8:0]}} \leftarrow \emptyset$
  end if
  $\tilde{h}^{\texttt{vec[u8]}} \leftarrow$ LENGTHENCODE 1($\tilde{i}$) ▷ III-A2.A
  $\tilde{k}^{\texttt{vec[u8]}} \leftarrow$ CONCATBYTEARRAYS ($\tilde{h}, \tilde{n}$) ▷ VIII-C2
  return $\tilde{k}$
end function

III-A10.B. IterateEncodeForSequence:

- Algorithm 5 encodes a sequence or a tuple i.e., data types `ary` (see Section II-B3.G) or `vec` (see Section II-B3.H) or `tup` (see Section II-B3.I), by iterating over its elements and returns a blob of encoded byte array i.e., `vec[u8]`.
- Algorithm 5 shall include its index of data types (see Section III-A3.A) in the encoded byte array.
- Test cases for Algorithm 5 are not required since Test Cases of SEQUENCEENCODE (see Section IX-A4) and TUPLEENCODE (see Section IX-A5) ensures the correctness of the ITERATEENCODEFORSEQUENCE Algorithm/Function.

III-A10.C. IterateValueEncodeForSequence:

- Algorithm 6 encodes a sequence i.e., data types `ary` (see Section II-B3.G) or `vec` (see Section II-B3.H), by iterating over its elements and returns a blob of encoded byte array i.e., `vec[u8]`.
- Algorithm 6 will not be including the elements index of data types (see Section III-A3.A) in the encoded byte array.
- Algorithm 6 should be strictly avoided for input of sequences S holding sequences of Option (see Section III-A4) or Result

Algorithm 5 IterateEncodeForSequence() $\mathcal{O}()$

function ITERATEENCODEFORSEQUENCE($S^{\texttt{vecVaryVtup}}$)
  for all $e_i \in S$ **do** ▷ II-B2
    if $\tilde{n}^{\texttt{vec[u8]}} = \emptyset$ **then**
      $\tilde{n}^{\texttt{vec[u8]}} \leftarrow$ ENCODE 10(e_i) ▷ III-A14.A
    else
      $\tilde{u}^{\texttt{vec[u8]}} \leftarrow$ ENCODE 10(e_i) ▷ III-A14.A
      $\tilde{n}^{\texttt{vec[u8]}} \leftarrow$ CONCATBYTEARRAYS ($\tilde{n}, \tilde{u}$) ▷ VIII-C2
    end if
  end for
  return $\tilde{n}$
end function

data types (see Section III-A5) of data type index $3 \geq i \geq 0$ (see Section III-A3.A)

- Test cases for Algorithm 6 are not required since Test Cases of SEQUENCEENCODE (see Section IX-A4) ensures the correctness of the ITERATEVALUEENCODEFORSEQUENCE Algorithm/Function.

Algorithm 6 IterateValueEncodeForSequence() $\mathcal{O}()$

function ITERATEVALUEENCODEFORSEQUENCE($S^{\texttt{vec} \vee \texttt{ary}}, k^{\texttt{u8}}$)
  for all $e_i \in S$ **do** ▷ II-B2
    if $\tilde{n}^{\texttt{vec[u8]}} = \emptyset$ **then**
      $\tilde{n}^{\texttt{vec[u8]}} \leftarrow$ VALUEENCODE 11(k, e_i) ▷ III-A14.B
    else
      $\tilde{u}^{\texttt{vec[u8]}} \leftarrow$ VALUEENCODE 11(k, e_i) ▷ III-A14.B
      $\tilde{n}^{\texttt{vec[u8]}} \leftarrow$ CONCATBYTEARRAYS ($\tilde{n}, \tilde{u}$) ▷ VIII-C2
    end if
  end for
  return $\tilde{n}$
end function

11) Tuple Encoding

A *Tuple*, T, is defined as a sequence variant where each A_i represents individual data types:

$$T = \{A_0, \ldots, A_i, \ldots, A_n\}$$

The encoding process of a tuple involves encoding the length of the tuple, denoted as LENGTHENCODE($|T|$) (see Algorithm 1), which is the number of elements in the tuple. This is followed by concatenating the individually encoded elements A_i in the sequence.

The Tuple type is a varying data type with index 9, i.e., $T_9(\mathcal{T})$ (see Section III-A3.A):

$$\text{TUPLEENCODE}(T) = \text{LENGTHENCODE}(9 \in T_9(\mathcal{T})) \parallel \text{LENGTHENCODE}(|T|) \parallel \text{ENCODE}(A_0) \parallel \ldots \parallel \text{ENCODE}(A_n)$$

Examples of Tuple Type are provided in Section A8.

III-A11.A. TupleEncode():

- Algorithm 7 encodes a tuple i.e., `tup` (see Section II-B3.I), and returns a blob of encoded byte array i.e., `vec[u8]`.
- Test cases for Algorithm 7 are provided in Section IX-A5.

Algorithm 7 TupleEncode() $\mathcal{O}()$

function TUPLEENCODE($S^{\texttt{tup}}$)
  $\tilde{i}^{\texttt{u-size}} \leftarrow$ NUMELEMENTS(S) ▷ VIII-A2
  if $\tilde{i} > 0$ **then**
    $\tilde{a}^{\texttt{vec[u8]}} \leftarrow$ ITERATEENCODEFORSEQUENCE 5(S) ▷ III-A10.B
  else
    $\tilde{a}^{\texttt{vec[u8]}} \leftarrow \emptyset$
  end if
  $\tilde{h}^{\texttt{vec[u8]}} \leftarrow$ LENGTHENCODE 1($\tilde{i}$) ▷ III-A2.A
  $\tilde{k}^{\texttt{vec[u8]}} \leftarrow$ CONCATBYTEARRAYS ($\tilde{h}, \tilde{a}$) ▷ VIII-C2
  return $\tilde{k}$
end function

12) String Encoding

A String is a collection of bytes consisting of UTF-8 encoded sequences (see Section II-A4). Since characters in UTF-8 encoding can vary in length but are self-describing, the string is represented as a byte array (`vec[u8]`). The encoding of a string type includes the length encoding (see Section III-A2) of the number of elements | `str` | i.e., individual UTF-8 elements, followed by converting the string's byte array to little-endian format.

The String type is a varying data type with index 6, i.e., $T_6(\mathcal{T})$ (see Section III-A3.A):

$$\text{ENCODE}(\texttt{str}) = \text{LENGTHENCODE}(6 \in T_6(\mathcal{T})) \parallel \text{LENGTHENCODE}(|\,\texttt{str}\,|) \parallel \text{LITTLEENDIAN}(\texttt{str})$$

Examples of String Type are provided in Section A9.

III-A12.A. StringEncode:

- Algorithm 8 encodes a string sequence i.e., `str` (see Section II-B3.F), and returns a blob of encoded byte array i.e., `vec[u8]`.
- Test cases for Algorithm 8 are provided in Section IX-A6.

Algorithm 8 StringEncode() $\mathcal{O}()$

function STRINGENCODE($S^{\texttt{str}}$)
  $\tilde{i}^{\texttt{u-size}} \leftarrow$ NUMELEMENTS(S) ▷ VIII-A2
  if $\tilde{i} > 0$ **then**
    $\tilde{k}^{\texttt{vec[u8]}} \leftarrow$ LEBYTEARRAY(S) ▷ VIII-B1
  else
    $\tilde{k}^{\texttt{vec[u8]}} \leftarrow \emptyset$
  end if
  $\tilde{h}^{\texttt{vec[u8]}} \leftarrow$ LENGTHENCODE 1($\tilde{i}$) ▷ III-A2.A
  $\tilde{v}^{\texttt{vec[u8]}} \leftarrow$ CONCATBYTEARRAYS ($\tilde{h}, \tilde{k}$) ▷ VIII-C2
  return $\tilde{v}$
end function

13) Special Data Types

Special Data Types provide encoding specifications for new higher-level data types or an alias, inheriting the attributes of fundamental data types $\{T_0, \ldots, T21\}(\mathcal{T})$ (see Section III-A3.A). These special data types must be:

1) Listed in the varying data type index (see Section III-A3.A) structure for ALAN.
2) Incorporated as a variant of FINDDATATYPEINDEX Algorithm/Function (see Algorithm 2).
   - In case the special data type inherits one of these fundamental data types $\{\texttt{vec}, \texttt{ary}, \texttt{tuple}\}$ attributes, the condition of index ensuring in Algorithm 2 should be included within its inherited data type's variant.
3) Covered in the test cases of FINDDATATYPEINDEX Algorithm/Function (see Section IX-A2).

4) Incorporated into the VALUEENCODE Algorithm/Function (see Algorithm 11).
   - Condition, and instructions included in the highest index of data type III-A3.A order, can be ignored if the Space or Time complexity of the VALUEENCODE Algorithm/Function (see Algorithm 11) can be avoided being incremented.
5) Addressed in the test cases of VALUEENCODE Algorithm/Function (see Section IX-A8).
6) Optional to Include in the SEQUENCEENCODE Algorithm/Function (see Algorithm 4), if required.
7) Optional to be covered in the test cases of SEQUENCEENCODE Algorithm/Function (see Section IX-A4), if Algorithm 4 is altered.

III-A13.A. Nibble Encoding:

Some data structures, such as state-trie keys (see Section **??**) in a Radix-trie, organize their structure based on each nibble (half-byte, i.e., 4 bits) as key for the key-value database **??**. A nibble `nib` data type can be constructed from an existing fundamental data type. If in an implementation-specific language, a `u4` type may be used for storing half-byte values, this section on nibble encoding can be omitted as it can be viewed as a fixed-length type (see Section III-A9), and special data type requirements can be constructed accordingly. If the `u4` type is not present, then the following specifications shall be utilized to the following special data types.

Two new varying data types are defined for nibbles:

- $\texttt{nib}^{\texttt{u8}}$, listed in the varying data type index (see Section III-A3.A) with index $T_{22}(\mathcal{T})$ (see Section III-A3), wraps a nibble of value $\texttt{u8} < 16$ into a `u8` value by restricting the most significant 4 bits.
- `vec[nib]`, listed in the varying data type index with index $T_{23}(\mathcal{T})$, defines a vector (see Section II-B3.H) of nibbles i.e., $\texttt{nib}^{\texttt{u8}}$ of variable length as a new data type.
- `ary[nib:N]`, listed in the varying data type index with index $T_{24}(\mathcal{T})$, defines an array (see Section II-B3.G) of nibbles i.e., $\texttt{nib}^{\texttt{u8}}$ of fixed length `N` as a new data type.

The nibble encoding function is defined in Algorithm 11 ($k = 22$) to encode a $\texttt{nib}^{\texttt{u8}}$ value, where $\texttt{u8} < 16$, and the encoded blob will be the little-endian encoding of its `u8` value (exact to its inherited `u8` type i.e., a fixed length type - see Section III-A9):

$$\textsc{Encode}(\texttt{nib}^{\texttt{u8}}) = \textsc{LittleEndian}(\texttt{nib}^{\texttt{u8}})$$

Example for a Nibble Encoding are given in A10.

The function for encoding a sequence of nibbles is defined in Algorithm 9 to encode a vector (see Section II-B3.H) or an array (see Section II-B3.G) of $\texttt{nib}^{\texttt{u8}}$ into a canonical byte array. The encoding process of a sequence of nibbles includes the length of the post-encoding number of nibbles followed by the encoding specification which depends on whether the number of nibbles in the sequence is even ($n \mod 2 = 0$) or odd ($n \mod 2 = 1$).

In a sequence of `nib` i.e., `vec[nib]` or `ary[nib:N]`,

- If the number of elements in the sequence $n = |S^{\texttt{ary[nib:N]}}|$ is even ($n \mod 2 = 0$), then each pair of consecutive nibbles is combined into a byte, where the first nibble is multiplied by 16, and added to the second nibble.
- If the number of elements in the nibble sequence is odd ($n \mod 2 = 1$), then the first nibble remains as is, and the subsequent pairs of nibbles are combined into bytes as in the even case.

$$k^{\texttt{bool}} \leftarrow \mid S \mid \mod 2$$

$$P \leftarrow \begin{cases} \text{if k = 0} & \begin{array}{c}(16 \cdot \texttt{nib}_0^{\texttt{u8}} + \texttt{nib}_1^{\texttt{u8}}), \parallel \\ \ldots \parallel (16 \cdot \texttt{nib}_{n-1}^{\texttt{u8}} + \texttt{nib}_n^{\texttt{u8}})\end{array} \\ \text{if k = 1} & \begin{array}{c}\texttt{nib}_0^{\texttt{u8}} \parallel (16 \cdot \texttt{nib}_1^{\texttt{u8}} + \texttt{nib}_2^{\texttt{u8}}) \parallel \\ \ldots \parallel (16 \cdot \texttt{nib}_{n-1}^{\texttt{u8}} + \texttt{nib}_n^{\texttt{u8}})\end{array} \end{cases}$$

- The encoding of a nibble sequence would be defined as

$$\textsc{NibbleEncode}(S^{\texttt{ary[nib:N]}}) = \textsc{LengthEncode}(\mid P \mid) \parallel \textsc{BoolEncode}(k) \parallel P$$

Examples for Nibble Sequence Encoding are given in A11.

III-A13.B. NibbleSequenceEncode():

- The Algorithm 9 encodes a sequence of nibbles i.e., `vec[nib]` or `ary[nib]` into a shortened encoded byte array.
- The Algorithm 9's *Test Cases* are given in IX-A7.

Algorithm 9 NibbleSequenceEncode() $\mathcal{O}()$

```
function NIBBLESEQUENCEENCODE(S^{vec ∨ ary})
    m̃^{u-size} ← NUMELEMENTS(S)                          ▷ VIII-A2
    if m̃ > 0 then
        concurrent 3
            ĩ^{type(m̃)} ← 0
            k̃^{type(m̃)} ← 0
            j̃^{bool} ← m̃ mod 2
        end concurrency
        if j̃ = true then
            while m̃ > ĩ ≥ 0 do
                if ĩ ∈ e_ĩ(S) = 0 then
                    u8_k̃(p̃^{vec[u8]}) ← e_ĩ
                else
                    u8_k̃(p̃^{vec[u8]}) ← ((16 × e_{ĩ−1}) + e_ĩ)
                end if
                concurrent 2
                    ĩ ← ĩ + 2
                    k̃ ← k̃ + 1
                end concurrency
            end while
        else
            while m̃ > ĩ ≥ 0 do
                u8_k̃(p̃^{vec[u8]}) ← ((16 × e_ĩ) + e_{ĩ+1})
                concurrent 2
                    ĩ ← ĩ + 2
                    k̃ ← k̃ + 1
                end concurrency
            end while
        end if
        t̃^{vec[u8]} ← BOOLENCODE 3(j̃)                    ▷ III-A7.A
        p̃^{vec[u8]} ← CONCATBYTEARRAYS (t̃, p̃)             ▷ VIII-C2
    else
        p̃^{vec[u8]} ← ∅
    end if
    ẽ^{vec[u8]} ← LENGTHENCODE 1(k̃)                       ▷ III-A2.A
    q̃^{vec[u8]} ← CONCATBYTEARRAYS (ẽ, p̃)                 ▷ VIII-C2
    return q̃
end function
```

14) Encode Entrypoint

III-A14.A. Encode():

- The Algorithm 10 receives a data type's value from which it determines the index of the data type from $\mathcal{T}$ III-A3.A, encodes the value, concatenates byte arrays, and returns the encoded blob of a bytes.

- The return type of function call inside the algorithm : FINDDATATYPEINDEX (see Section III-A3.B) is kept `u8` since the current highest index of the index of data types (see Section III-A3.A) is $n < 256$.
- Test cases for Algorithm 10 are not required since it only include function calls to other test case ensured functions. Hence Algorithm 10's correctness is ensured via its nested functions test cases.

Algorithm 10 Encode() $\mathcal{O}()$

function ENCODE(n)
  $\tilde{k}^{\texttt{u8}} \leftarrow$ FINDDATATYPEINDEX 2(n) ▷ III-A3.B
  concurrent 2
  $\tilde{h}^{\texttt{vec[u8]}} \leftarrow$ LENGTHENCODE 1($\tilde{k}$) ▷ III-A2.A
  $\tilde{v}^{\texttt{vec[u8]}} \leftarrow$ VALUEENCODE 11($\tilde{k}, n$) ▷ III-A14.B
  end concurrency
  $\tilde{n}^{\texttt{vec[u8]}} \leftarrow$ CONCATBYTEARRAYS($\tilde{h}, \tilde{v}$) ▷ VIII-C2
   return $\tilde{n}$
end function

III-A14.B. ValueEncode():

- The Algorithm 11 encodes a data type's value based on its data type index III-A3.A provided in its input parameter.
- The input type of $k^{\texttt{u8}}$ is kept `u8` since the current highest index of the index of data types (see Section III-A3.A) is $n < 256$.
- The index of data types are evaluated from higher index to lower index, since special data types shall inherit the fundamental data types.
- The Algorithm 11's *Test Cases* given in IX-A8

Algorithm 11 ValueEncode() $\mathcal{O}()$

function VALUEENCODE($k^{\texttt{u8}}, n$)
  if $24 \geq k \geq 23$ **then** ▷ III-A13.A
    $\texttt{u8}_0(\tilde{v}^{\texttt{vec[u8]}}) \leftarrow$ NIBBLESEQUENCEENCODE 9(n) ▷ III-A13.B
  else if $22 \geq \tilde{k} \geq 10$ **then** ▷ III-A9
    $\tilde{n}^{\texttt{vec[u8]}} \leftarrow$LEBYTEARRAY($n$) ▷ VIII-B1
  else if $k = 9$ **then** ▷ III-A11
    $\tilde{v}^{\texttt{vec[u8]}} \leftarrow$ TUPLEENCODE 7(n) ▷ III-A11.A
  else if $8 \geq k \geq 7$ **then** ▷ III-A10,III-A12
    $\tilde{v}^{\texttt{vec[u8]}} \leftarrow$ SEQUENCEENCODE 4(n) ▷ III-A10.A
  else if $k = 6$ **then** ▷ III-A12
    $\tilde{v}^{\texttt{vec[u8]}} \leftarrow$ STRINGENCODE 8(n) ▷ III-A12.A
  else if $k = 5$ **then** ▷ III-A8
    $\tilde{v}^{\texttt{vec[u8]}} \leftarrow$ LEBYTEARRAY(n) ▷ VIII-B1
  else if $k = 4$ **then** ▷ III-A7
    $\tilde{v}^{\texttt{vec[u8]}} \leftarrow$ BOOLENCODE 3(n) ▷ III-A7.A
  else if $3 \geq k \geq 1$ **then** ▷ III-A4.B,III-A5
    $\tilde{v}^{\texttt{vec[u8]}} \leftarrow$ ENCODE 10(n) ▷ III-A14.A
  else if $k = 0$ **then** ▷ III-A4.A
    $\tilde{v}^{\texttt{vec[u8]}} \leftarrow$ LENGTHENCODE 1(0) ▷ III-A2.A
  end if
   return $\tilde{v}$
end function

B. Decoding

The blob of encoded data can be deserialized into the given type or data structure.

IV. HOST

A. State Database

Key-Value Encoded Database for Storage & Retrieval of Runtime V written Data

B. Actions

Authenticated Messages to Invoke Runtime

C. Reserve

To Evaluate & Store Actions for Streaming with Bartergas

D. Snips

Validation & Dissemination of Stream Author Executed Actions

E. Stream

Authenticated Ordered Set of Snips with fixed Stream Number

F. Permits

For Authenticating Actions's Origin

G. Authors

Authors assigned for upcoming Stream Number

H. Host's Runtime

Host Methods to Invoke Runtime V-A

I. Units

Compute-Space-Time Unit of Alan

J. P2P (Peer to Peer)

Peer-to-Peer Networking using QUIC Protocol

K. Commits

Making Snips Permanent in Alan

L. Recovery

Recovering from Snip Tampering Attacks by Ring-Stream authors

M. GPU Access

Web-GPU Access to Host Functions

V. RUNTIME

A. Invoke Runtime

Entrypoint for Actions Executing Runtime Functions, Invoke Runtime via Runtime

B. State Access

Access to State Database IV-A by Runtime via Host IV

C. Wasm Function Trie

Instantiation, Caching, Runtime Upgrades & Requirements

D. Function Variants & Owners

Runtime Functions Version Control & Ownership Rules

E. Frame Custodians

Custodians of transient wasm stack-frames a.k.a owner of a runtime function that creates those temporary stack-frames

F. Author Auction

Author Selection Procedure assigned for Stream IV-E numbers

G. Publish To Reserve

Runtime access to write to Host's Reserves IV-C

H. Permit Entries

Deposits for turing-incomplete permit evaluation scripts.

I. GPU Access

Web-GPU Access to Runtime Functions

VI. RUNTIME ADDONS

1) Wasm Programs Module

VII. NORMS & BEST PRACTICES

Standards, and Instructions for best desired outcomes from Run-time Implementations - E.g., BarterGas Contracts

VIII. SUPPLEMENTARY FUNCTIONS

This supplementary section includes descriptions or specifications of feature functions of programming languages, categorized into specific operations. These functions can be called by multiple functions across ALAN's specifications. Implementations may choose to ignore or include these logic depending on the availability of the necessary feature in the programming language or its standard libraries. It is advised to ensure that the feature functions take the exact parameters of the specified data types, and provide the output in the desired data type.

A. Length Functions

1) LenBytes()

LENBYTES($n^{\texttt{type}}$) returns a `u-size` data type value which represents the number of bytes occupied by the data type $n^{\texttt{type}}$ in memory. This function helps determine the memory footprint of a given type, providing how much space it consumes.

2) NumElements()

NUMELEMENTS($n^{\texttt{vecVaryVtup}}$) returns a `u-size` data type value which represents the number of elements in the sequence $n^{\texttt{vecVaryVtup}}$. The number of elements is known at compile time for data types such as arrays (see Section II-B3.G) and tuples (see Section II-B3.I). However, for dynamic sequences like vectors (see Section II-B3.H), the length is determined at runtime.

B. Type Conversions

1) LeByteArray()

LEBYTEARRAY($n^{\texttt{type}}$) returns a vector sequence (see Section II-B3.H) of `u8` elements by encoding the input type's bytes in little-endian order. This function is used to convert a value of any type into a vector of bytes with a specific byte order, useful for data serialization (see Section III-A).

C. Little Endian Byte Array Functions

1) ShiftLeByteArrayBitToMsb()

- The Algorithm takes a little-endian encoded byte array, `vec[u8]`, and shifts its bits to the most significant bit position indicated by its input k, where the maximum shift can be $k < 2^8$.
- The Algorithm 's *Test Cases* are given in IX-C0.A.

2) ConcatByteArrays()

CONCATBYTEARRAYS($a^{\texttt{vec[u8]}}, b^{\texttt{vec[u8]}}$) function takes two vector byte array sequences as inputs and concatenates them in order. This means the elements of the second vector are appended to the elements of the first vector, resulting in a new combined vector.

Algorithm 12 ShiftLeByteArrayBitToMsb() $\mathcal{O}(n)$

function SHIFTLEBYTEARRAYBITTOMSB($n^{\texttt{vec[u8]}}, k^{\texttt{u8}}$)
  $\tilde{k}^{\texttt{u-size}} \leftarrow$ NUMELEMENTS(n)
  if $\tilde{k} > 0$ **then**
    if $k > 0$ **then**
      $\tilde{u}^{\texttt{u8}} \leftarrow \lceil \frac{k}{8} \rceil$
      concurrent 2
      $\tilde{i_b}^{\texttt{u8}} \leftarrow k \mod 8$
      $\tilde{i_B}^{\texttt{u8}} \leftarrow \lfloor \frac{k}{8} \rfloor$
      end concurrency
      for all $\texttt{u8}_i \in n$ **do**
        for all $b_i \in \texttt{u8}_i(n)$ **do**
          $b_{i_b}(\texttt{u8}_{i_B}(\tilde{p}^{\texttt{vec[u8]}})) \leftarrow b_i(\texttt{u8}_i(n))$
          $b_i(\texttt{u8}_i(\tilde{p})) \leftarrow 0^{\texttt{bool}}$
          $\tilde{i_b} = (\tilde{i_b} + 1) \mod 8$
          if $\tilde{i_b} = 0$ **then**
            $\tilde{i_B} = \tilde{i_B} + 1$
          end if
        end for
      end for
    else
      $\tilde{p}^{\texttt{vec[u8]}} \leftarrow n$
    end if
  else
    $\tilde{p}^{\texttt{vec[u8]}} \leftarrow \emptyset$
  end if
   return $\tilde{p}$
end function

IX. TESTING SPECIFICATION (CASES)

Inputs, and expected output are provided in square brackets [] as Base-16 (Hexadecimal) representations (see II-A3).

If any higher-level programming language's specific data type does not accept a zero or an empty value, the empty test case can be omitted during implementation.

A. Encoding Functions

1) LengthEncode

The Test Cases are given for LENGTHENCODE 1 function in Section III-A2.A.

1) Small Range Case ($2^0 \leq n < 2^6$)
   a) `u8` value
      - *Input* : $n^{\texttt{u8}} = [32]$
      - *Exp.Output* : $k^{\texttt{vec[u8]}} = [C8_0]$
   b) `u16` value
      - *Input* : $n^{\texttt{u16}} = [001E]$
      - *Exp.Output* : $k^{\texttt{vec[u8]}} = [78_0]$
   c) `u32` value
      - *Input* : $n^{\texttt{u32}} = [0000001A]$
      - *Exp.Output* : $k^{\texttt{vec[u8]}} = [68_0]$
   d) `u64` value
      - *Input* : $n^{\texttt{u64}} = [000000000000000F]$
      - *Exp.Output* : $k^{\texttt{vec[u8]}} = [3C_0]$
   e) `u128` value
      - *Input* : $n^{\texttt{u128}} = [0000000000000000000000000000003C]$
      - *Exp.Output* : $k^{\texttt{vec[u8]}} = [F0_0]$
2) Medium Range Case ($2^6 \leq n < 2^{14}$)
   a) `u8` value
      - *Input* : $n^{\texttt{u8}} = [84]$
      - *Exp.Output* : $k^{\texttt{vec[u8]}} = [11_0, 02_1]$
   b) `u16` value
      - *Input* : $n^{\texttt{u16}} = [03E8]$
      - *Exp.Output* : $k^{\texttt{vec[u8]}} = [A1_0, 0F_1]$
   c) `u32` value
      - *Input* : $n^{\texttt{u32}} = [000036CE]$
      - *Exp.Output* : $k^{\texttt{vec[u8]}} = [39_0, DB_1]$
   d) `u64` value
      - *Input* : $n^{\texttt{u64}} = [0000000000003CA0]$
      - *Exp.Output* : $k^{\texttt{vec[u8]}} = [81_0, F2_1]$
   e) `u128` value
      - *Input* : $n^{\texttt{u128}} = [00000000000000000000000000003F48]$
      - *Exp.Output* : $k^{\texttt{vec[u8]}} = [21_0, FD_1]$
3) Large Value Case ($2^{14} \leq n < 2^{30}$)
   a) `u16` value
      - *Input* : $n^{\texttt{u16}} = [7D28]$
      - *Exp.Output* : $k^{\texttt{vec[u8]}} = [A2_0, F4_1, 01_2, 00_3]$
   b) `u32` value
      - *Input* : $n^{\texttt{u32}} = [000F5758]$
      - *Exp.Output* : $k^{\texttt{vec[u8]}} = [62_0, 5D_1, 3D_2, 00_3]$
   c) `u64` value
      - *Input* : $n^{\texttt{u64}} = [000000000BF36334]$
      - *Exp.Output* : $k^{\texttt{vec[u8]}} = [D2_0, 8C_1, CD_2, 2F_3]$
   d) `u128` value
      - *Input* : $n^{\texttt{u128}} = [0000000000000000000000001FF6DAE3]$
      - *Exp.Output* : $k^{\texttt{vec[u8]}} = [8E_0, 6B_1, DB_2, 7F_3]$
4) Maximum Value Case ($> n \geq 2^{30}$)
   a) `u32` value
      - *Input* : $n^{\texttt{u32}} = [7D2BEC24]$
      - *Exp.Output* :

$$k^{\texttt{vec[u8]}} = [13_0, 24_1, EC_2, 2B_3, 7D_4]$$

   b) `u64` value
      - *Input* : $n^{\texttt{u64}} = [0DF27A5B88562710]$
      - *Exp.Output* :

$$k^{\texttt{vec[u8]}} = [23_0, 10_1, 27_2, 56_3, 88_4, 5B_5, 7A_6, F2_7, 0D_8]$$

   c) `u128` value
      - *Input* :

$$n^{\texttt{u128}} = [94471DBEA19A0E3DF6EB78A03D2A5740]$$

      - *Exp.Output* :

$$k^{\texttt{vec[u8]}} = [43_0, 40_1, 57_2, 2A_3, 3D_4, A0_5, 78_6, EB_7, F6_8, 3D_9, 0E_{10}, 9A_{11}, A1_{12}, BE_{13}, 1D_{14}, 47_{15}, 94_{16}]$$

2) FindDataTypeIndex

The Test Cases are given for FINDDATATYPEINDEX 2 function in Section III-A3.B.

1) `u8` Type
   - *Input* : $n^{\texttt{u8}} = [8A]$
   - *Exp.Output* : $k^{\texttt{u8}} = [0A]$
2) `u16` Type
   - *Input* : $n^{\texttt{u16}} = [058A]$
   - *Exp.Output* : $k^{\texttt{u8}} = [0B]$
3) `u32` Type
   - *Input* : $n^{\texttt{u32}} = [AB2D058A]$
   - *Exp.Output* : $k^{\texttt{u8}} = [0C]$
4) `u64` Type
   - *Input* : $n^{\texttt{u64}} = [AB2D058AAB2D058A]$
   - *Exp.Output* : $k^{\texttt{u8}} = [0D]$
5) `u128` Type
   - *Input* :

$$n^{\texttt{u128}} = [AB2D058AAB2D058AAB2D058AAB2D058A]$$

   - *Exp.Output* : $k^{\texttt{u8}} = [0E]$
6) `i8` Type
   - *Input* : $n^{\texttt{i8}} = [8A]$
   - *Exp.Output* : $k^{\texttt{u8}} = [0F]$
7) `i16` Type
   - *Input* : $n^{\texttt{i16}} = [058A]$
   - *Exp.Output* : $k^{\texttt{u8}} = [10]$
8) `i32` Type
   - *Input* : $n^{\texttt{i32}} = [AB2D058A]$
   - *Exp.Output* : $k^{\texttt{u8}} = [11]$
9) `i64` Type
   - *Input* : $n^{\texttt{i64}} = [AB2D058AAB2D058A]$
   - *Exp.Output* : $k^{\texttt{u8}} = [12]$
10) `i128` Type
   - *Input* :

$$n^{\texttt{i128}} = [AB2D058AAB2D058AAB2D058AAB2D058A]$$

   - *Exp.Output* : $k^{\texttt{u8}} = [13]$
11) `f32` Type
   - *Input* : $n^{\texttt{f32}} = [AB2D058A]$
   - *Exp.Output* : $k^{\texttt{u8}} = [14]$

12) `f64` Type
   - *Input* : $n^{\texttt{f64}} = [\text{AB2D058AAB2D058A}]$
   - *Exp.Output* : $k^{\texttt{u8}} = [15]$
13) `bool` Type
   - *Input* : $n^{\texttt{bool}} = \text{true}$
   - *Exp.Output* : $k^{\texttt{u8}} = [04]$
14) `char` Type
   - *Input* : $n^{\texttt{char}} = \text{"A"}$
   - *Exp.Output* : $k^{\texttt{u8}} = [05]$
15) `str` Type
   - *Input* : $n^{\texttt{str}} = \text{"Hello"}$
   - *Exp.Output* : $k^{\texttt{u8}} = [06]$
16) `ary` Type
   - *Input* : $n^{\texttt{ary[u8:3]}} = [\text{A1}_0, 65_1, \text{D2}_2]$
   - *Exp.Output* : $k^{\texttt{u8}} = [08]$
17) `vec` Type
   - *Input* : $n^{\texttt{vec[u8]}} = [\text{A1}_0, 65_1, \text{D2}_2, \text{FF}_3]$
   - *Exp.Output* : $k^{\texttt{u8}} = [07]$
18) `tup` Type
   - *Input* : $n^{\texttt{tup[u8,char]}} = [\text{A1}_0, \text{"A"}_1]$
   - *Exp.Output* : $k^{\texttt{u8}} = [09]$
19) Option Type
   a) `none` Type
      - *Input* : $n^{\texttt{none:i}} = []$
      - *Exp.Output* : $k^{\texttt{u8}} = [00]$
   b) `some` Type
      - *Input* : $n^{\texttt{some:u8}} = [\text{A4}]$
      - *Exp.Output* : $k^{\texttt{u8}} = [01]$
20) Result Type
   a) `Ok` Type
      - *Input* : $n^{\texttt{Ok:u8}} = [\text{7D}]$
      - *Exp.Output* : $k^{\texttt{u8}} = [02]$
   b) `Err` Type
      - *Input* : $n^{\texttt{Err:u8}} = [\text{8C}]$
      - *Exp.Output* : $k^{\texttt{u8}} = [03]$
21) Nibble Type
   - *Input* : $n^{\texttt{nib}} = [\text{A5}]$
   - *Exp.Output* : $k^{\texttt{u8}} = [16]$
22) Nibble Vector Type
   - *Input* : $n^{\texttt{vec[nib]}} = [\text{A5}_0, \text{7C}_0]$
   - *Exp.Output* : $k^{\texttt{u8}} = [17]$
23) Nibble Array Type
   - *Input* : $n^{\texttt{ary[nib:2]}} = [\text{A5}_0, \text{7C}_0]$
   - *Exp.Output* : $k^{\texttt{u8}} = [18]$

3) BoolEncode

The Test Cases are given for BOOLENCODE 3 function in Section III-A7.A.

1) True Case
   - *Input* : $n^{\texttt{bool}} = \texttt{true}$
   - *Exp.Output* : $v^{\texttt{vec}[u8]} = [01_0]$
2) False Case
   - *Input* : $n^{\texttt{bool}} = \texttt{false}$
   - *Exp.Output* : $v^{\texttt{vec}[u8]} = [00_0]$

4) SequenceEncode

The Test Cases are given for SEQUENCEENCODE 4 function in Section III-A10.A.

1) Array
   a) Empty `u8` Array
      - *Input* : $n^{\texttt{ary[u8:0]}} = []$
      - *Exp.Output* : $v^{\texttt{vec[u8]}} = [00_0]$
   b) Result Type Array
      i) `u8,i8` Result Type Array
         - *Input* :
           $$n^{\texttt{ary[[[Ok:u8],[Err:i8]]:2]}} = [(\texttt{Ok} : \text{AF}), (\texttt{Err} : \text{F8})]$$
         - *Exp.Output* :
           $$v^{\texttt{vec[u8]}} = [08_0, 08_1, 28_2, \text{AF}_3, \text{0C}_4, \text{3C}_5, \text{F8}_6]$$
      ii) `u64,str` Result Type Array
         - *Input* :
           $$n^{\texttt{ary[[[Ok:u64],[Err:str]]:2]}} = [(\texttt{Ok} : \text{7CFCA468A85CAB82}), (\texttt{Err} : \text{"Hello"})]$$
         - *Exp.Output* :
           $$v^{\texttt{vec[u8]}} = [08_0, 08_1, 34_2, 82_3, \text{AB}_4, \text{5C}_5, \text{A8}_6, 68_7, \text{A4}_8, \text{FC}_9, \text{7C}_{10}, \text{0C}_{11}, 18_{12}, 14_{13}, \text{6F}_{14}, \text{6C}_{15}, \text{6C}_{16}, 65_{17}, 48_{18}]$$
      iii) `f32,char` Result Type Array
         - *Input* :
           $$n^{\texttt{ary[[[Ok:f32],[Err:char]]:2]}} = [(\texttt{Ok} : \text{7CFCAB82}), (\texttt{Err} : \text{"A"})]$$
         - *Exp.Output* :
           $$v^{\texttt{vec[u8]}} = [08_0, 08_1, 50_2, 82_3, \text{AB}_4, \text{FC}_5, \text{7C}_6, \text{0C}_7, 14_8, 41_9, 00_{10}, 00_{11}, 00_{12}]$$
      iv) `bool,nib` Result Type Array
         - *Input* :
           $$n^{\texttt{ary[[[Ok:bool],[Err:nib]]:3]}} = [(\texttt{Ok} : \texttt{true}), (\texttt{Err} : 06), (\texttt{Err} : \text{0A})]$$
         - *Exp.Output* :
           $$v^{\texttt{vec[u8]}} = [\text{0C}_0, 08_1, 10_2, 01_3, \text{0C}_4, 58_5, 06_6, \text{0C}_7, 58_8, \text{0A}_9]$$
   c) Option Type Array
      i) `none,i8` Option Type Array
         - *Input* :
           $$n^{\texttt{ary[[[none],[some:i8]]:2]}} = [(\texttt{none}, (\texttt{some} : \text{F7}))]$$
         - *Exp.Output* :
           $$v^{\texttt{vec[u8]}} = [08_0, 00_1, 04_2, \text{3C}_3, \text{F7}_4]$$
      ii) `none,str` Option Type Array
         - *Input* :
           $$n^{\texttt{ary[[[none],[some:str]]:2]}} = [(\texttt{none}, (\texttt{some} : \text{"Hello"}))]$$
         - *Exp.Output* :
           $$v^{\texttt{vec[u8]}} = [08_0, 00_1, 04_2, 18_3, 14_4, \text{6F}_5, \text{6C}_6, \text{6C}_7, 65_8, 48_9]$$
      iii) `none,char` Option Type Array

- *Input* :

$$n^{\texttt{ary[[none],[some:char]]:2]}} = [(\texttt{none}, (\texttt{some} : \text{"A"}))]$$

- *Exp.Output* :

$$v^{\texttt{vec[u8]}} = [08_0, 00_1, 04_2, 14_3, 41_4, 00_5, 00_6, 00_7]$$

iv) `none`,`nib` Option Type Array

- *Input* :

$$n^{\texttt{ary[[none],[some:nib]]:3]}} = [(\texttt{none}, (\texttt{some} : 0F), (\texttt{some} : 02))]$$

- *Exp.Output* :

$$v^{\texttt{vec[u8]}} = [0C_0, 00_1, 04_2, 58_3, 0F_4, 04_5, 58_6, 02_7]$$

d) `u8` Array

- *Input* :

$$n^{\texttt{ary[u8:3]}} = [98_0, FA_1, 6A_2]$$

- *Exp.Output* :

$$v^{\texttt{vec[u8]}} = [0C_0, 28_1, 98_0, FA_2, 6A_3]$$

e) `i16` Array

- *Input* :

$$n^{\texttt{ary[i16:1]}} = [A6FB_0]$$

- *Exp.Output* :

$$v^{\texttt{vec[u8]}} = [04_0, 40_1, FB_0, A6_2]$$

f) `f64` Array

- *Input* :

$$n^{\texttt{ary[f64:2]}} = [28514A12A7A5BD5A_0, 37A90E4551D3A8AA_1]$$

- *Exp.Output* :

$$v^{\texttt{vec[u8]}} = [08_0, 54_1, 5A_2, BD_3, A5_4, A7_5, 12_6, 4A_7, 51_8, 28_9, AA_{10}, A8_{11}, D3_{12}, 51_{13}, 45_{14}, 0E_{15}, A9_{16}, 37_{17}]$$

g) `char` Array

- *Input* :

$$n^{\texttt{ary[char:2]}} = [\text{"A"}_0, \text{"B"}_1]$$

- *Exp.Output* :

$$v^{\texttt{vec[u8]}} = [08_0, 14_1, 41_2, 00_3, 00_4, 00_5, 42_6, 00_7, 00_8, 00_9]$$

h) `bool` Array

- *Input* :

$$n^{\texttt{ary[bool:3]}} = [\texttt{true}_0, \texttt{true}_1, \texttt{false}_2]$$

- *Exp.Output* :

$$v^{\texttt{vec[u8]}} = [0C_0, 10_1, 01_2, 01_3, 00_4]$$

i) `str` Array

- *Input* :

$$n^{\texttt{ary[str:2]}} = [\text{"Hello"}_0, \text{"Bob"}_1]$$

- *Exp.Output* :

$$v^{\texttt{vec[u8]}} = [08_0, 18_1, 14_2, 6F_3, 6C_4, 6C_5, 65_6, 48_7, 0C_8, 62_9, 6F_{10}, 42_{11}]$$

j) `nib` Array

- *Input* :

$$n^{\texttt{ary[nib:2]}} = [05_0, 0B_1]$$

- *Exp.Output* :

$$v^{\texttt{vec[u8]}} = [08_0, 58_1, 05_2, 0B_3]$$

k) `vec[nib]` Array

- *Input* :

$$n^{\texttt{ary[vec[nib]:2]}} = [([05_0, 0B_1]), ([0A_0, 03_1, 02_2])]$$

- *Exp.Output* :

$$v^{\texttt{vec[u8]}} = [08_0, 5C_1, 04_2, 00_3, 5B_4, 08_5, 01_6, 0A_7, 32_8]$$

l) `ary[nib:2]` Array

- *Input* :

$$n^{\texttt{ary[ary[nib:2]:2]}} = [([05_0, 0B_1]), ([0A_0, 03_1])]$$

- *Exp.Output* :

$$v^{\texttt{vec[u8]}} = [08_0, 60_1, 04_2, 00_3, 5B_4, 04_5, 00_6, A3_7]$$

2) Vector : Test cases derived from Array can be tested for Vectors where *Input* : $n^{\texttt{vec[i]}}$.

5) TupleEncode

The Test Cases are given for TUPLEENCODE 7 function in Section III-A11.A.

1) Empty Tuple Type

- *Input* :

$$n^{\texttt{tup[]}} = []$$

- *Exp.Output* :

$$v^{\texttt{vec[u8]}} = [00_0]$$

2) Fixed Length Types Tuple

- *Input* :

$$n^{\texttt{tup[(u8),(i16),(f32)]}} = [(8B), (F6BA), (F78A34BD)]$$

- *Exp.Output* :

$$v^{\texttt{vec[u8]}} = [0C_0, 28_1, 8B_2, 40_3, BA_4, F6_5, 50_6, BD_7, 34_8, 8A_9, F7_{10}]$$

3) String `nib` Type & Char `char` Type Tuple

- *Input* :

$$n^{\texttt{tup[(str),(char)]}} = [(\text{"Hi"}), (\text{"K"})]$$

- *Exp.Output* :

$$v^{\texttt{vec[u8]}} = [08_0, 18_1, 08_2, 69_3, 48_4, 14_5, 4B_6, 00_7, 00_8, 00_9]$$

4) Tuple of : Vector Type of `u16` of length 2 & Array Type of `bool` of length 2

- *Input* :

$$n^{\texttt{tup[(vec[u8]),(ary[bool:2])]}} = [([(7B), (9F)]), ([(\texttt{true}), (\texttt{false})])]$$

- *Exp.Output* :

$$v^{\texttt{vec[u8]}} = [08_0, 1C_1, 08_2, 28_3, 7B_4, 9F_5, 20_6, 08_7, 10_8, 01_9, 00_{10}]$$

5) Tuple of : Vector of `nib` of length 3 & Nibble `nib` Type

- *Input* :

$$n^{\texttt{tup[(vec[nib]),(nib)]}} = [([(0B), (0F), (05)]), (0A)]$$

- *Exp.Output* :

$$v^{\texttt{vec[u8]}} = [08_0, 5C_1, 08_2, 01_3, 0B_4, F5_5, 58_6, 0A_7]$$

6) Tuple of : Array of `nib` of length 2 & Nibble `nib` Type
   - *Input* :

$$n^{\texttt{tup[(ary[nib:2]),(nib)]}} = [([(0B),(0F)]),(0A)]$$

   - *Exp.Output* :

$$v^{\texttt{vec[u8]}} = [08_0, 60_1, 04_2, 00_3, BF_4, 58_5, 0A_6]$$

6) StringEncode

The Test Cases are given for STRINGENCODE 8 function in Section III-A12.A.

1) Empty String Type
   - *Input* : $n^{\texttt{str}} = ""$
   - *Exp.Output* : $v^{\texttt{vec[u8]}} = [00_0]$
2) Single Length String
   - *Input* : $n^{\texttt{str}} = "A"$
   - *Exp.Output* : $v^{\texttt{vec[u8]}} = [04_0, 41_1]$
3) Normal Case
   - *Input* : $n^{\texttt{str}} = "Alan"$
   - *Exp.Output* : $v^{\texttt{vec[u8]}} = [10_0, 6E_1, 61_2, 6C_3, 41_4]$

7) NibbleSequenceEncode

The Test Cases are given for NIBBLESEQUENCEENCODE 9 function in Section III-A13.B.

1) Vector Case
   a) Empty Case
      - *Input* : $n^{\texttt{vec[nib]}} = []$
      - *Exp.Output* : $v^{\texttt{vec[nib]}} = [00]$
   b) Single Nibble Case
      - *Input* : $n^{\texttt{vec[nib]}} = [05_0]$
      - *Exp.Output* : $v^{\texttt{vec[nib]}} = [04_0, 01_1, 05_2]$
   c) Odd Case
      - *Input* : $n^{\texttt{vec[nib]}} = [04_0, 0D_1, 0A_2]$
      - *Exp.Output* : $v^{\texttt{vec[nib]}} = [08_0, 01_1, 04_2, DA_3]$
   d) Even Case
      - *Input* : $n^{\texttt{vec[nib]}} = [06_0, 0C_1, 0F_2, 03_3]$
      - *Exp.Output* : $v^{\texttt{vec[nib]}} = [08_0, 00_1, 6C_2, F3_3]$
2) Array Case : Test cases derived from Vector can be tested for Nibble Arrays where *Input* : $n^{\texttt{ary[nib:N]}}$

8) ValueEncode

The Test Cases are given for VALUEENCODE 11 function in Section III-A14.B. Test cases are derived from several test cases such as

1) Derive Test Cases from NIBBLESEQUENCE (see Section IX-A7) where, the *Input* k is,
   - $k^{\texttt{u8}} = 23$ if $\texttt{type} \in n^{\texttt{type[nib]}} = \texttt{vec}$
   - $k^{\texttt{u8}} = 24$ if $\texttt{type} \in n^{\texttt{type[nib]}} = \texttt{ary}$
2) Derive Test Cases from TUPLEENCODE (see Section IX-A5) where, the *Input* : $k^{\texttt{u8}} = 9$
3) Derive Test Cases from SEQUENCEENCODE (see Section IX-A4) where, the *Input* k is,
   - $k^{\texttt{u8}} = 7$ if $\texttt{type} \in n^{\texttt{type}} = \texttt{vec}$
   - $k^{\texttt{u8}} = 8$ if $\texttt{type} \in n^{\texttt{type}} = \texttt{ary}$
4) Derive Test Cases from STRINGENCODE (see Section IX-A6) where, the *Input* : $k^{\texttt{u8}} = 6$
5) Derive Test Cases from BOOLENCODE (see Section IX-A3) where, the *Input* : $k^{\texttt{u8}} = 4$
6) For `some` Type, `Ok` Type, and `Err` Type, Derive Test Cases from,
   a) NIBBLESEQUENCE (see Section IX-A7)
   b) TUPLEENCODE (see Section IX-A5)
   c) SEQUENCEENCODE (see Section IX-A4)
   d) BOOLENCODE (see Section IX-A3)
7) `none` Type, Derive *Zero or Empty Length* Individual Test Cases from,
   a) NIBBLESEQUENCE (see Section IX-A7)
   b) TUPLEENCODE (see Section IX-A5)
   c) SEQUENCEENCODE (see Section IX-A4)
   d) BOOLENCODE (see Section IX-A3)

B. Decoding Functions

C. Supplementary Functions

IX-C0.A. ShiftLeByteArrayBitToMsb:

The Test Cases are given for SHIFTLEBYTEARRAYBITTOMSB function in Section VIII-C1.

1) Empty Byte Array
   - *Input* : $n^{\texttt{vec[u8]}}, k^{\texttt{u8}} = [], [05]$
   - *Exp.Output* : $i^{\texttt{vec[u8]}} = []$
2) Zero Bit Shift
   - *Input* : $n^{\texttt{vec[u8]}}, k^{\texttt{u8}} = [32_0, AF_1], [00]$
   - *Exp.Output* : $i^{\texttt{vec[u8]}} = [32_0, AF_1]$
3) Single Bit Shift
   - *Input* : $n^{\texttt{vec[u8]}}, k^{\texttt{u8}} = [32_0, AF_1], [01]$
   - *Exp.Output* : $i^{\texttt{vec[u8]}} = [64_0, 5E_1, 01_2]$
4) Odd Bit Shift
   - *Input* : $n^{\texttt{vec[u8]}}, k^{\texttt{u8}} = [32_0, AF_1], [03]$
   - *Exp.Output* : $i^{\texttt{vec[u8]}} = [20_0, F3_1, 0A_2]$
5) 8 Bit Shift
   - *Input* : $n^{\texttt{vec[u8]}}, k^{\texttt{u8}} = [32_0, AF_1], [08]$
   - *Exp.Output* : $i^{\texttt{vec[u8]}} = [00_0, 32_1, AF_2]$

Appendix

A. Specification Examples

This section includes all extended examples that provide pseudo-values to help understand topics that are covered in Alan's specifications.

1) LittleEndian

The Examples are given for LITTLEENDIAN function defined in Section III-A1.

Lets take a non-negative integer of value $k = 8500000$

The binary (`base2`) value of k is

$$k = \{10000001_0 \ 10110011_1 \ 00100000_2\}$$

or $k = 0\text{x}\{81_0 \ \text{B}3_1 \ 20_2\}$ in `base-16` hex representation

The little-endian byte order is achieved by reversing the order of the bytes (but not reversing the bits within each byte). This reversal is because, in little-endian order, the least significant byte comes first.

$$\{00100000_0 \ 10110011_1 \ 10000001_2\} \text{ or } \{20_0 \ \text{B}3_1 \ 81_2\}_{\text{base-16}}$$

Alternatively, we can define in `base-256`, $\mathbb{Z}(8500000) = \{129_0, 179_1, 32_2\}$ where $(32\times256^0)+(179\times256^1)+(129\times256^2) = 8500000$

Hence the *Little Endian* of $8500000_{\mathbb{Z}} = \{32_0, 179_1, 129_2\}$

2) Length Encoding

The Examples are given for LENGTHENCODE function (see Algorithm 1) defined in Section III-A2.

For **Case 1 (1 Byte)** where $0 \le n < 2^6$ i.e., $0 \le n < 63$

Lets take a non-negative integer of value $n = 50$

- Binary Representation = 110010_0 (6 bits)
- Complete Byte Representation = 00110010_0 (8 bits)
- Little Endian Encoding = 00110010_0 (8 bits)
- Since Little Endian acts on byte array the representation for 1 byte is the same
- Shifting Bits to Right by 2 bits = 11001000_0 (8 bits)
- Replacing Least Significant Bits $(b_1^0, b_0^0) = (0, 0) = 11001000_0$ (8 bits)
- Since 8 bits = 1 Byte for Case 1

Hence

$$11001000_0 \leftarrow \text{LENGTHENCODE}(50)$$

Proof:

- $(b_1^0, b_0^0) = (0, 0)$ as least significant bits (LSb) of the least significant byte (LSB) representing 1-byte format length ($\mathbb{B}_1$).
- Hence, $n = (b_7^0 \cdot 2^5) + \ldots + (b_5^0 \cdot 2^0), b_1^0, b_0^0$
- Where b_n^m represents

$$\text{bit}^{\text{byte number}}_{\text{bit number}}$$

- The byte (see Section II-A1) is of the structure

$$\mathbb{B}_n = (B_0, B_1, \ldots B_{n-1})$$

- The bit number (see Section II-A2) is of the structure

$$B = (b_7, \ldots, b_0)$$

- With $11001000 \leftarrow \text{LENGTHENCODE}(50)$, the integer will be $2^5 + 2^4 + 2^1 = 50 = n$ represented in 1 Byte

For **Case 2 (2 Byte)** where $2^6 \le n < 2^{14}$ i.e., $64 \le n < 16383$

Lets take a non-negative integer of value $n = 10000$

- Binary Representation = $100111_0 \ 00010000_1$ (14 Bits)
- Complete Byte Representation = $00100111_0 \ 00010000_1$ (16 Bits)
- Little Endian Encoding = $00100111_1 \ 00010000_0$ (16 bits)
- Shifting Bits to Right by 2 bits = $10011100 \ 01000000_0$ (16 bits)
- Replacing Least Significant Bits $(b_1^0, b_0^0) = (0, 1) = 1001110001000001$ (16 bits)
- Since 16 bits = 2 Bytes for Case 2

Hence $10011100_1 \ 01000001_0 \leftarrow \text{LENGTHENCODE}(10000)$

Proof:

- $(b_1^0, b_0^0) = (0, 1)$ as least significant bits (LSb) of the least significant byte (LSB) representing 2-byte format length ($\mathbb{B}_2$).
- Hence, $n = (b_7^0 \cdot 2^5) + \ldots + (b_2^0 \cdot 2^0) + (b_7^1 \cdot 2^{13}) + \ldots (b_0^1 \cdot 2^6)$
- With $10011100_1 \ 01000001_0 \leftarrow \text{LENGTHENCODE}(10000)$, the integer will be $2^4+2^8+2^9+2^{10}+2^{13} = 10000 = n$ represented in two bytes

For **Case 3 (4 Byte)** where $2^{14} \le n < 2^{30}$ i.e., $16384 \le n < 1073741823$ is similar to **Case 2** with additional two bytes

For **Case 4 ($m+1$ Bytes)** where $n \ge 2^{30}$ i.e., $n \ge 1073741824$

Lets take a non-negative integer with value $n = 10000000000$

- Binary Representation = 1001010100000010111110010000000000 (34 Bits)
- Complete Byte Representation (40 Bits) =

$$00000010_0 \ 01010100_1 \ 00001011_2 \ 11100100_3 \ 00000000_4$$

- Little Ending Encoding (40 Bits) =

$$00000010_4 \ 01010100_3 \ 00001011_2 \ 11100100_1 \ 00000000_0$$

- Find the Length of Original Integer (Total Number of Bytes) = 34 Bits = $\frac{2^{34}}{\text{8bits per byte}} = \lceil 4.25 \rceil = 5$. Hence $m = 5$
- Total Bytes to Represent the n is $m + 1 = 6$ is 6 bytes
- Extra Byte = 00010111, where $b_1, b_0 = 1, 1$ is the LSb representing the case of variable integer $n \ge 2^{30}$, and rest $000101 = 2^0 + 2^2 = 5$ is the length of original integer m
- Extra byte as Least Significant Byte = $00000010_5 \ 01010100_4 \ 00001011_3 \ 11100100_2 \ 00000001_1 \ \| \ 00010111_0$ (48 bits)

Hence

$$00000010_5 \ 01010100_4 \ 00001011_3 \ 11100100_2 \ 00000001_1$$
$$00010111_0 \leftarrow \text{LENGTHENCODE}(10000000000)$$

Proof:

- $(b_1^0, b_0^0) = (1, 1)$ as least significant bytes representing $m + 1$ byte format length ($\mathbb{B}_{m+1}$) including the LSb, where $\mathbb{B}_0$ is the Extra Byte, and $\mathbb{B}_{m+1} \ldots \mathbb{B}_1$ represent the integer information
- $\{b_7^0, \ldots, b_2^0\} = 000101_{base2} = 5_{base10}$, this provides that the total bytes will be 5 i.e., based on zeroth-index is $5 - 1 = 4$ from $\mathbb{B}_4, \ldots, \mathbb{B}_0$
- So we take the next 5 bytes $00000010_4 \ 01010100_3 \ 00001011_2 \ 11100100_1 \ 00000000_0$ from which we can derive its base256 $2 \cdot 256^4 + 84 \cdot 256^3 + 11 \cdot 256^2 + 228 \cdot 256^1 + 0 \cdot 256^0 = 8589934592 + 1409286144 + 720896 + 58368 = 10000000000$

3) Varying Data Type

Let's take $A = \{A_{\text{type}}, A_{\text{value}}\} = \{\texttt{u8}, 175\}$. For this varying data type, we know that the index of `u8` data type is 10 according to Index of Data Types (see Section III-A3.A), the encoding would be

$$\text{ENCODE}(A) = \text{LENGTHENCODE}(10 \in T_{10}(\mathcal{T})) \ \| \ \text{ENCODE}(\texttt{u8} = 175)$$

where, $\{00101000_0\} \leftarrow \text{LENGTHENCODE}(10)$. Hence,

$$\text{ENCODE}(A) = \{00101000_0\} \ \| \ \text{ENCODE}(\texttt{u8} = 175)$$

4) Some Type

The encoding for the Some type (a type with value present) i.e., from Index of Data Types (see Section III-A3.A) a `u8` type with index $i = 10 \in T_i(\mathcal{T})$ with value 175 i.e., $A = \{A_{\text{type}}, A_{\text{value}}\} = \{\texttt{u8}, 175\}$ would be,

$$\textsc{Encode}(A) = \textsc{LengthEncode}(1 \in T_1(\mathcal{T})) \parallel \textsc{LengthEncode}(10 \in T_{10}(\mathcal{T})) \parallel \textsc{Encode}(\texttt{u8} = 500)$$

where,

- LENGTHENCODE (see Section III-A2.A & Algorithm 1) is used.
- $1 \in T_1(\mathcal{T}) =$ `Some` because `Some` is the second element in the ordered set $\mathcal{T}$ (see Section III-A3.A) which has a zero-based indexing.
- $\{00000100_0\} \leftarrow \textsc{LengthEncode}(1)$

Hence the result would be

$$\textsc{Encode}(\texttt{u8}, 500) = \{00000100_0\} \parallel \textsc{LengthEncode}(10 \in T_{10}(\mathcal{T})) \parallel \textsc{Encode}(\texttt{u8} = 500)$$

5) Result Type

The encoding for the Result type of `Ok` would be, $\textsc{Encode}(2 \in T_2(\mathcal{T}), A_{\text{type}} A_{\text{value}})$ according to Index of Data Types (see Section III-A3.A)

Lets take an example of $\textsc{Encode}(\texttt{Ok}, \texttt{u8}, 500)$. The encoding scheme will be similar to Some Type Example (see Section A4),

$$\textsc{Encode}(\texttt{Ok}, \texttt{u8}, 500) = \textsc{LengthEncode}(2) \parallel \textsc{LengthEncode}(4) \parallel \textsc{Encode}(\texttt{u8} = 500)$$

The encoding result would be,

$$\textsc{Encode}(\texttt{Ok}, \texttt{u8}, 500) = \{00001000_0\} \parallel \textsc{LengthEncode}(4) \parallel \textsc{Encode}(\texttt{u8} = 500)$$

The encoding for the Result type of `Err` would be, $\textsc{Encode}(3 \in T_2(\mathcal{T}), A_{\text{type}} A_{\text{value}})$ according to Index of Data Types (see Section III-A3.A)

Lets take an example of $\textsc{Encode}(\texttt{Err}, \texttt{u8}, 500)$. The encoding scheme will be similar to Some Type Example (see Section A4),

$$\textsc{Encode}(\texttt{Err}, \texttt{u8}, 500) = \textsc{LengthEncode}(3) \parallel \textsc{LengthEncode}(4) \parallel \textsc{Encode}(\texttt{u8} = 500)$$

The encoding result would be,

$$\textsc{Encode}(\texttt{Err}, \texttt{u8}, 500) = \{00001100_0\} \parallel \textsc{LengthEncode}(4) \parallel \textsc{Encode}(\texttt{u8} = 500)$$

6) Character Encoding

Since a character is a UTF-8 Encoded value which has a maximum length of 4 bytes, The character encoding involves conversion of a `char` value to a little endian byte array

Let's take an example of a character in UTF-8 : *Latin Capital Letter A With Tilde* in double quotes "$\tilde{A}$" the unicode decimal is given as 195, and its unicode byte of length 4 bytes (in big endian format) is $\{0000000_0\ 0000000_1\ 10000011_2\ 11000011_3\}$

The encoding of a character `char` of *Latin Capital Letter A With Tilde* in double quotes "$\tilde{A}$" will be

$$\textsc{Encode}(\texttt{char} = \tilde{A}) = \textsc{LengthEncode}(5) \parallel \textsc{LittleEndian}(0000000_0\ 0000000 willbe_1\ 10000011_2\ 11000011_3)$$

where, $\textsc{LengthEncode}(5)$ represents `char`'s index according to Index of Data Types (see Section III-A3.A)

The encoding result will be,

$$\textsc{Encode}(\texttt{char} = \tilde{A}) = \{00010100_0\ 11000011_1\ 10000011_2\ 00000000_3\ 00000000_4\}$$

7) Sequence Encoding

The encoding for sequence type which involves `vec,ary`

1) For an example of an `u8` array $\textsc{Encode}(\texttt{ary}[\texttt{u8} : 2])$, the encoding would be,

$$\textsc{Encode}(\texttt{ary}[\texttt{u8} : 2]) = \textsc{LengthEncode}(8) \parallel \textsc{LengthEncode}(2) \parallel \textsc{LengthEncode}(10) \parallel \textsc{ValueEncode}(\texttt{u8}_0) \ldots \parallel \textsc{ValueEncode}(\texttt{u8}_1)\text{III-A14.}B$$

where,
- $\textsc{LengthEncode}(8)$: Represents `array`'s index of varying data type (see Section III-A3.A)
- $\textsc{LengthEncode}(2)$: Represents two elements in the array
- $\textsc{LengthEncode}(10)$: Represents `u8`'s index of varying data type (see Section III-A3.A)
- $\textsc{ValueEncode}()$ (see Section III-A14.B): Encodes only the value without involving the data type index, as sequences will have same type elements, it is only necessary to include its first element's data type.

2) For an example of an `u8` vector $\textsc{Encode}(\texttt{vec}[\texttt{u8}])$, the encoding would be,

$$\textsc{Encode}(\texttt{vec}[\texttt{u8}]) = \textsc{LengthEncode}(7) \parallel \textsc{LengthEncode}(\mid \texttt{vec} \mid) \parallel \textsc{LengthEncode}(10) \parallel \textsc{ValueEncode}(\texttt{u8}_0) \ldots \parallel \textsc{ValueEncode}(\texttt{u8}_{(|\texttt{vec}|-1)})$$

3) For elements of sequences of data types `Result` (see Section III-A5), the encoding would be,

$$\textsc{Encode}(\texttt{vec}[(\texttt{Ok:u8}), (\texttt{Err:i16})]) = \textsc{LengthEncode}(7) \parallel \textsc{LengthEncode}(2) \parallel \textsc{Encode}(\texttt{Ok} \vee \texttt{Err}_0) \parallel \ldots \parallel \textsc{Encode}(\texttt{Ok} \vee \texttt{Err}_n)$$

where,
- Instead of the first element's data type, all the elements are subjected to individual encoding similar to tuple encoding (see Section III-A11)

4) For elements of sequences of data types `Option` (see Section III-A4), the encoding would be,

$$\textsc{Encode}(\texttt{vec}[(\texttt{none}), (\texttt{some:i16})]) = \textsc{LengthEncode}(7) \parallel \textsc{LengthEncode}(2) \parallel \textsc{Encode}(\texttt{none} \vee \texttt{some}_0) \parallel \ldots \parallel \textsc{Encode}(\texttt{none} \vee \texttt{some}_n)$$

8) Tuple Encoding

Since tuples have multiple different type sequences, every individual element shall be encoded individually, whereas the number of elements is similar to sequence encoding.

Lets take an example of an `u8,u16,u32` tuple $\textsc{Encode}(\texttt{tup}[\texttt{u8,u16,u32}])$ would be encoded as follows:

$$\textsc{Encode}(\texttt{tup}) = \textsc{LengthEncode}(9) \parallel \textsc{LengthEncode}(\mid \texttt{tup} \mid) \parallel \textsc{Encode}(\texttt{u8}_0) \parallel \textsc{Encode}(\texttt{u16}_1) \parallel \textsc{Encode}(\texttt{u32}_2)$$

where,

- $\textsc{LengthEncode}(9)$ represents `tuple`'s varying data type index (see Section III-A3.A)
- $\mid \texttt{tup} \mid$ represents the length of the tuple, i.e., the number of elements in the tuple. In the above example, it is three.

9) String Encoding

A string is a byte array that includes a sequence of UTF-8 elements. Since, the UTF-8 bytes self describes lengths of each element, it requires to be encoded in little endian order.

Lets Take an example string "Hello" for which the big endian UTF-8 byte sequence will be $\{01001000_0\ 01100101_1\ 01101100_2\ 01101100_3\ 01101111_4\}$

The encoding of string "Hello" is defined as,

$$\textsc{Encode}(\texttt{str} = \text{"Hello"}) = \textsc{LengthEncode}(6) \parallel \textsc{LittleEndian}(01001000_0\ 01100101_1\ 01101100_2\ 01101100_3\ 01101111_4)$$

where, $\textsc{LengthEncode}(6)$ represents `str`'s index $T_6(\mathcal{T})$ according to Index of Data Types (see Section III-A3.A)

The encoding result will be,

$$\textsc{Encode}(\texttt{str} = \text{"Hello"}) = \{00011000_0\ 01101111_1\ 01101100_2\ 01101100_3\ 01100101_4\ 01001000_5\}$$

10) Nibble Encoding

Lets take a nibble of value $\texttt{nib}^{\texttt{u8}} = 00000110$ where each `nib` is a `u8` value of $\texttt{u8} < 16$

The encoding result of $\texttt{nib}^{\texttt{u8}} = 00000110$ would be

$$\textsc{Encode}(\texttt{nib}^{\texttt{u8}} = 00000110) = \textsc{LengthEncode}(22) \parallel 00000110$$

11) Nibble Sequence Encoding

Lets take an odd nibble sequence of length 3 i.e., number of elements $\{00001010_0\ 00001101_1\ 00000011_2\}$ where according to $\texttt{nib}^{\texttt{u8}}$ (see Section A10) each nibble is a `u8` value

The encoding result of the odd nibble vector sequence would be

$$\textsc{NibbleSequenceEncode}(\texttt{vec[nib]} = \{00001010_0\ 00001101_1\ 00000011_2\}) = \textsc{LengthEncode}(23) \parallel \textsc{LengthEncode}(2) \parallel 00001010_0 \parallel (16 \times 00001101_1 + 00000011_2)$$

where,

- $\textsc{LengthEncode}(23)$: Represents `vec[nib]`'s index of varying data type (see Section III-A3.A)
- Here, according to the Nibble Sequence Encoding specification (see Section III-A13.A), for odd case the first nibble $\texttt{nib}_0$ remains as is, and the subsequent pairs of nibbles are combined into bytes
- $\textsc{LengthEncode}(2)$: Represents total number of bytes in the post-encoded byte array.

Lets take an even nibble sequence of length 4 : $\{00001010_0\ 00001101_1\ 00000011_2\ 00001101_3\}$.

The encoding result of an even nibble vector sequence would be

$$\textsc{NibbleSequenceEncode}(\texttt{vec[u8]} = \{00001010_0\ 00001101_1\ 00000011_2\ 00001101_3\}) = \textsc{LengthEncode}(23) \parallel \textsc{LengthEncode}(2) \parallel (16_{\texttt{base10}} \times 00001010_0 + 00001101_1) \parallel (16_{\texttt{base10}} \times 00000011_2 + 00001101_3)$$

Similarly for nibble array sequence i.e., `ary[nib:N]` the index of varying data type (see Section III-A3.A) only changes.

B. Contribution Formats

1) Section

2) Algorithms

3) Test-Cases

4) Labels & Reference

C. GNU Free Documentation License

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1) Applicability and Definitions

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "**Document**", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "**you**". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "**Modified Version**" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "**Secondary Section**" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "**Invariant Sections**" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "**Cover Texts**" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "**Transparent**" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "**Opaque**".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "**Title Page**" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "**publisher**" means any person or entity that distributes copies of the Document to the public.

A section "**Entitled XYZ**" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "**Acknowledgements**", "**Dedications**", "**Endorsements**", or "**History**".) To "**Preserve the Title**" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2) Verbatim Copying

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3) Copying in Quantity

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4) Modifications

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
D. Preserve all the copyright notices of the Document.
E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
H. Include an unaltered copy of this License.
I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5) Combining Documents

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number.

Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6) Collections of Documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7) Aggregation with Independent Works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8) Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9) Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10) Future Revisions of the License

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `https://www.gnu.org/licenses/`.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11) Relicensing

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

D. Version Logs

Version ID	Version Name	Fingerprint	PDF Link	Source Link	Contributor Name	Contributor Email
0.1	Draft	000000_{16}			Joby Reuben	*joby@auguth.org*