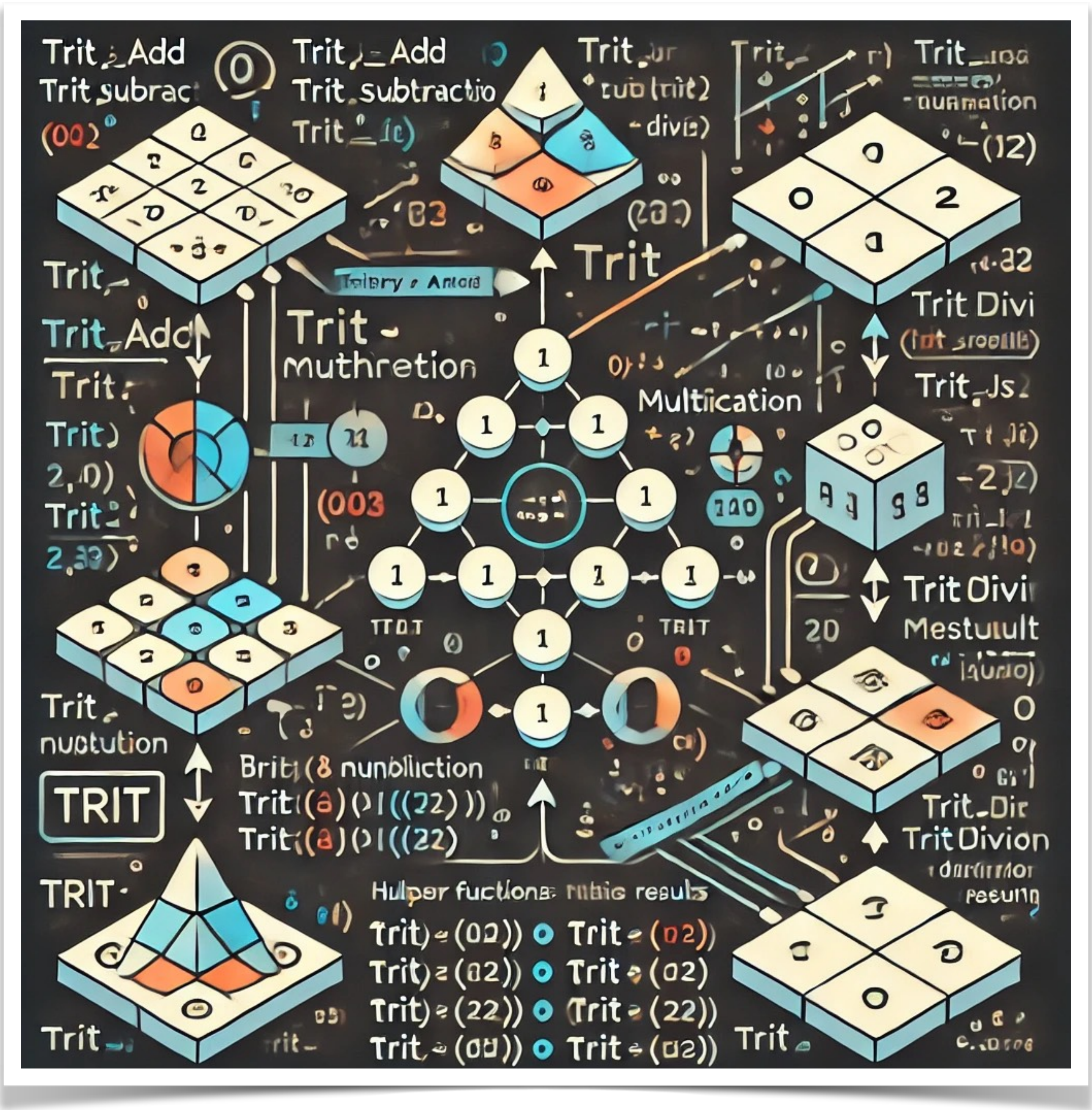
Ternary Arithmetic Library (Base-3) in C

"A custom library for addition, subtraction, multiplication, and division of ternary numbers; (trits: 0, 1, 2)."


Trit_Add
Trit subtraction
Trit
Multification
Trit Divi
Trit Division
TRIT
Helper functions

Section Details

1. **Data Structures**

    - **Visual**:
        - A box labeled `Trit` with "0, 1, 2" inside, colored accordingly.
          A structure diagram for `TritDivResult`:
          ```
          TritDivResult
          ├─ quotient (Trit*) → [T, T, T]
          ├─ remainder (Trit*) → [T, T]
          ├─ q_len (int)
          └─ r_len (int)
          ```
        - Caption: "Trit = single digit (0-2); Arrays for numbers."

    Code Snippet: c
    ```
    #define TRIT_MAX 3
    typedef int Trit;
    typedef struct { Trit* quotient; Trit* remainder;
    int q_len; int r_len; } TritDivResult;
    ```

2. Helper Functions

- **Visual**:
  - Two sub-boxes:
    - **Conversion**: Arrow from `[1, 2]` (trits) to `5` (binary) and back.
    - **Single-Trit Math**:
      - Addition: `1 + 2 = {value: 0, carry: 1}`
      - Subtraction: `1 - 2 = {value: 2, borrow: 1}`
      - Multiplication: `2 * 2 = {value: 1, carry: 1}`
  - Caption: "Utilities for conversion and trit-level arithmetic."

Code Snippet: c

```
unsigned long trits_to_binary(Trit* trits, int len);
Trit* binary_to_trits(unsigned long bin, int len);
TritSum trit_add(Trit a, Trit b);
```

3. Core Operations

- **Visual**:
  - A flowchart for each operation:
    - **Addition**: `[1, 2] + [2, 1]` → `[1, 1, 0]` (carry shown in red).
    - **Subtraction**: `[1, 2] - [2, 1]` → `[2]` (borrow in red).
    - **Multiplication**: `[1, 2] * [2, 1]` → `[1, 0, 1]`.
    - **Division**: `[1, 2] / [2, 1]` → `quotient: [0, 2]`, `remainder: [0, 1]`.
  - Caption: "Full ternary number operations with carry/borrow handling."

Code Snippet: c

```
Trit* tritjs_add(Trit* a, int a_len, Trit* b, int
b_len, int* result_len);
Trit* tritjs_multiply(Trit* a, int a_len, Trit*
b, int b_len, int* result_len);
TritDivResult tritjs_divide(Trit* a, int a_len,
Trit* b, int b_len);
```

4. Utility Functions

- **Visual**:
  - A string icon: `[1, 2] → "12"`.
  - A binary switch icon: "Native Ternary? No (0)".
  - Caption: "Debugging and system compatibility tools."
- **Code Snippet**: c

```c
char* tritjs_to_string(Trit* trits, int len);
int tritjs_is_ternary_native(void); /* Returns 0
*/
```

5. Example Usage

- **Visual**:
  - A table showing inputs and outputs:

Operation	Input A	Input B	Result
Add	12_3	21_3	110_3
Subtract	12_3	21_3	2_3
Multiply	12_3	21_3	101_3
Divide	12_3	21_3	02_3 r 01_3

  - Caption: "Test cases from main() function."

Code Snippet: c

```c
int main() {
    Trit a[] = {1, 2}; Trit b[] = {2, 1};
    Trit* sum = tritjs_add(a, 2, b, 2, &len);
    printf("Add: %s\n", tritjs_to_string(sum,
len));
}
```

Additional Elements

- **Legend**: Bottom-left corner with color meanings (e.g., Blue = 0, Red = Carry/Borrow).

- **Footer**: "Created for Alexis Linux | Base-3 Arithmetic | Feb 28, 2025."

- **Icons**:
  - Trits as small circles with numbers.
  - Arrows for data flow.
  - CPU icon next to "Native Ternary" to indicate hardware context.

"Trits → Binary”

Code Reference: c

```
unsigned long trits_to_binary(Trit* trits, int len) {
    unsigned long bin = 0;
    for (int i = 0; i < len; i++) {
        bin = (bin << 2) | trits[i]; /* Each trit
takes 2 bits */
    }
    return bin;
}
```

- **Example Input**: `[1, 2]` (representing $12_3 = 5_{10}$)
- **Visualization**:
  1. **Initial State**:
     - Box: `bin = 0` (binary: `0000`)
     - Array: `[1, 2]` (trits in green and orange).
  2. **Step 1 (i = 0)**:
     - Shift left 2 bits: `bin << 2` → `0000` becomes `0000`.
     - OR with trit[0]: `0000 | 1` → `0001` (green 1).
     - Caption: "Append trit 1 (uses 2 bits: 01)”.
  3. **Step 2 (i = 1)**:
     - Shift left 2 bits: `0001 << 2` → `0100`.
     - OR with trit[1]: `0100 | 2` → `0110` (orange 2 = binary 10).
     - Caption: "Append trit 2 (uses 2 bits: 10)”.
  4. **Final Result**:
     - Box: `bin = 0110` (binary) = 6_{10}.
     - Note: "Each trit takes 2 bits; result is shifted left.”
- **Diagram**:

```
[1, 2] →  bin: 0000
```

```
          Step 1: (<< 2) | 1 → 0001
          Step 2: (<< 2) | 2 → 0110
Output: 0110 (6₁₀)
```

"Binary → Trits”

Code Reference: `c`

```
Trit* binary_to_trits(unsigned long bin, int len) {
    Trit* trits = (Trit*)malloc(len * sizeof(Trit));
    for (int i = len - 1; i >= 0; i--) {
        trits[i] = bin & 0x3; /* Extract lowest 2 bits
*/
        bin >>= 2;
    }
    return trits;
}
```

- **Example Input**: `0110` (binary) with `len = 2`
- **Visualization**:
  1. **Initial State**:
     - Box: `bin = 0110` (binary).
     - Array: `[_, _]` (empty trits, len = 2).
  2. **Step 1 (i = 1)**:
     - AND with 0x3: `0110 & 0011` → `0010` (2 in orange).
     - Assign: `trits[1] = 2`.
     - Shift right 2 bits: `0110 >> 2` → `0001`.
     - Array: `[_, 2]`.
     - Caption: "Extract lowest 2 bits (10 = 2)”.
  3. **Step 2 (i = 0)**:
     - AND with 0x3: `0001 & 0011` → `0001` (1 in green).
     - Assign: `trits[0] = 1`.
     - Shift right 2 bits: `0001 >> 2` → `0000`.
     - Array: `[1, 2]`.
     - Caption: "Extract next 2 bits (01 = 1)”.

4. **Final Result**:
   - Box: `trits = [1, 2]` (representing 12_3).
   - Note: "Right shift discards processed bits."

- **Diagram**:

```
0110 →  trits: [_, _]
     Step 1: (0110 & 0x3) = 2, >> 2 → [_, 2], bin =
     0001
     Step 2: (0001 & 0x3) = 1, >> 2 → [1, 2], bin =
     0000
Output: [1, 2] (12₃)
```

Additional Elements

- **Comparison Arrow**:
  - Double-headed arrow between `[1, 2]` and `0110` with text: "Reversible Process".

- **Bit Representation**:
  - Show each trit as a 2-bit pair (e.g., 0 = 00, 1 = 01, 2 = 10) in a small table:

```
Trit | Binary
-----|-------
0    | 00
1    | 01
2    | 10
```

- **Caption**: "Trits use 2 bits each in binary; conversions preserve ternary value."

Design Notes

- **Flow Arrows**: Use curved arrows to show bit shifting (left for `trits_to_binary`, right for `binary_to_trits`).
- **Highlight**: Color-code the active bits/trits in each step (e.g., green for 1, orange for 2).
- **Tool Suggestion**: Create this in a vector graphics tool (e.g., Inkscape) or programmatically with Python's Matplotlib for step-by-step frames.

Example Summary Visualization

```
Trits to Binary: [1, 2]
  0000 → (<< 2 | 1) → 0001 → (<< 2 | 2) → 0110

Binary to Trits: 0110
  [_, _] → (0110 & 0x3 = 2, >> 2) → [_, 2] → (0001 & 0x3 =
1, >> 2) → [1, 2]
```

This visualization clearly shows the bitwise operations and array manipulation, making it easy to understand the conversion logic.

```
@c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

@*1 Data Structures and Constants.
We define a trit as an integer (0, 1, or 2) and use arrays to represent ternary numbers.
For convenience, we also define a structure for division results.

```
@d TRIT_MAX 3 /* Base-3 modulus */
 typedef int Trit; /* A single trit: 0, 1, or 2 */
 typedef struct {
    Trit* quotient; /* Array of trits */
    Trit* remainder; /* Array of trits */
    int q_len; /* Length of quotient */
    int r_len; /* Length of remainder */
} TritDivResult;
```

@*1 Helper Functions.
These utilities handle conversions and single-trit arithmetic.

```
@<Conversion Functions@>
@<Arithmetic Helpers@>
```

@ Conversions between trits and binary are crucial for division and debugging.

```
@c
@<Conversion Functions@>=
unsigned long trits_to_binary(Trit* trits, int len) {
    unsigned long bin = 0;
    for (int i = 0; i < len; i++) {
        bin = (bin << 2) | trits[i]; /* Each trit takes 2 bits */
    }
    return bin;
}

Trit* binary_to_trits(unsigned long bin, int len) {
    Trit* trits = (Trit*)malloc(len * sizeof(Trit));
    for (int i = len - 1; i >= 0; i--) {
        trits[i] = bin & 0x3; /* Extract lowest 2 bits */
        bin >>= 2;
    }
    return trits;
```

```
}
```

@ Single-trit arithmetic operations manage carries and borrows.

```
@c
@<Arithmetic Helpers@>=
typedef struct { Trit value; int carry; } TritSum;
TritSum trit_add(Trit a, Trit b) {
    int sum = a + b;
    return (TritSum){ sum % TRIT_MAX, sum / TRIT_MAX };
}

typedef struct { Trit value; int borrow; } TritDiff;
TritDiff trit_subtract(Trit a, Trit b) {
    int diff = a - b;
    if (diff >= 0) return (TritDiff){ diff, 0 };
    return (TritDiff){ (diff + TRIT_MAX) % TRIT_MAX, 1 };
}

typedef struct { Trit value; int carry; } TritProd;
TritProd trit_multiply(Trit a, Trit b) {
    int prod = a * b;
    return (TritProd){ prod % TRIT_MAX, prod / TRIT_MAX };
}
```

@*1 Core Arithmetic Operations.
Now we implement the main functions: addition, subtraction, multiplication, and division.

```
@c
Trit* tritjs_add(Trit* a, int a_len, Trit* b, int b_len, int* result_len) {
    int max_len = (a_len > b_len) ? a_len : b_len;
    Trit* result = (Trit*)malloc((max_len + 1) * sizeof(Trit)); /* Room for carry */
    int carry = 0, pos = 0;

    for (int i = max_len - 1; i >= 0; i--) {
        Trit a_trit = (i < a_len) ? a[i] : 0;
        Trit b_trit = (i < b_len) ? b[i] : 0;
        TritSum sum = trit_add(a_trit + carry, b_trit);
        result[max_len - pos] = sum.value;
        carry = sum.carry;
        pos++;
    }
    if (carry) {
```

```
        result[0] = carry;
        *result_len = max_len + 1;
    } else {
        memmove(result, result + 1, max_len * sizeof(Trit));
        *result_len = max_len;
    }
    return result;
}

Trit* tritjs_subtract(Trit* a, int a_len, Trit* b, int b_len, int* result_len) {
    int max_len = (a_len > b_len) ? a_len : b_len;
    Trit* result = (Trit*)malloc(max_len * sizeof(Trit));
    int borrow = 0;

    for (int i = max_len - 1; i >= 0; i--) {
        Trit a_trit = (i < a_len) ? a[i] : 0;
        Trit b_trit = (i < b_len) ? b[i] : 0;
        TritDiff diff = trit_subtract(a_trit - borrow, b_trit);
        result[i] = diff.value;
        borrow = diff.borrow;
    }
    /* Trim leading zeros */
    int start = 0;
    while (start < max_len - 1 && result[start] == 0) start++;
    *result_len = max_len - start;
    Trit* trimmed = (Trit*)malloc(*result_len * sizeof(Trit));
    memcpy(trimmed, result + start, *result_len * sizeof(Trit));
    free(result);
    return trimmed;
}

Trit* tritjs_multiply(Trit* a, int a_len, Trit* b, int b_len, int* result_len) {
    int max_len = a_len + b_len;
    Trit* result = (Trit*)calloc(max_len, sizeof(Trit));
    for (int i = a_len - 1; i >= 0; i--) {
        int carry = 0;
        for (int j = b_len - 1; j >= 0; j--) {
            int pos = i + j + 1;
            TritProd prod = trit_multiply(a[i], b[j]);
            TritSum sum = trit_add(result[pos] + carry, prod.value);
            result[pos] = sum.value;
            carry = sum.carry + prod.carry;
        }
```

```
        if (carry) {
            int carry_pos = i;
            TritSum sum = trit_add(result[carry_pos] + carry, 0);
            result[carry_pos] = sum.value;
            if (sum.carry) {
                memmove(result, result + 1, max_len * sizeof(Trit));
                result[0] = sum.carry;
                max_len++;
            }
        }
    }
    /* Trim leading zeros */
    int start = 0;
    while (start < max_len - 1 && result[start] == 0) start++;
    *result_len = max_len - start;
    Trit* trimmed = (Trit*)malloc(*result_len * sizeof(Trit));
    memcpy(trimmed, result + start, *result_len * sizeof(Trit));
    free(result);
    return trimmed;
}

TritDivResult tritjs_divide(Trit* a, int a_len, Trit* b, int b_len) {
    unsigned long a_bin = trits_to_binary(a, a_len);
    unsigned long b_bin = trits_to_binary(b, b_len);
    if (b_bin == 0) {
        fprintf(stderr, "Error: Division by zero\n");
        exit(1);
    }
    unsigned long quotient_bin = a_bin / b_bin;
    unsigned long remainder_bin = a_bin % b_bin;
    TritDivResult result = {
        .quotient = binary_to_trits(quotient_bin, a_len),
        .remainder = binary_to_trits(remainder_bin, b_len),
        .q_len = a_len,
        .r_len = b_len
    };
    return result;
}
```

@*1 Utility Functions.
We provide a string conversion for debugging and a placeholder for native ternary support.

```
@c
char* tritjs_to_string(Trit* trits, int len) {
    char* str = (char*)malloc(len + 1);
    for (int i = 0; i < len; i++) str[i] = '0' + trits[i];
    str[len] = '\0';
    return str;
}

int tritjs_is_ternary_native(void) {
    return 0; /* No native ternary support in standard C environments */
}
```

@*1 Example Usage.
Here’s a simple main function to test the library.

```
@c
int main() {
    Trit a[] = {1, 2}; /* 12₃ */
    Trit b[] = {2, 1}; /* 21₃ */
    int len;
    Trit* sum = tritjs_add(a, 2, b, 2, &len);
    printf("Add: %s\n", tritjs_to_string(sum, len)); /* Should print "110" */
    free(sum);

    Trit* diff = tritjs_subtract(a, 2, b, 2, &len);
    printf("Subtract: %s\n", tritjs_to_string(diff, len)); /* Should print "2" */
    free(diff);

    Trit* prod = tritjs_multiply(a, 2, b, 2, &len);
    printf("Multiply: %s\n", tritjs_to_string(prod, len)); /* Should print "101" */
    free(prod);

    TritDivResult div = tritjs_divide(a, 2, b, 2);
    printf("Divide: %s r %s\n", tritjs_to_string(div.quotient, div.q_len),
           tritjs_to_string(div.remainder, div.r_len)); /* Should print "02 r 01" */
    free(div.quotient);
    free(div.remainder);

    return 0;
}
```