BenO Compression Algorithm 2025-09

v0.2

Generated by Doxygen 1.12.0

1 Ben O Compression algorithm for interview 2025-09-19

1.1 Code Design Test: Data Compression Design

Design an algorithm that will compress a given data buffer of bytes.
Please describe your design and submit an implementation in a language of your choice.

- The algorithm will live within a function.
- This function will be called with two arguments;
  - A pointer to the data buffer (data_ptr)
  - The number of bytes to compress (data_size).
- After the function executes the data in the buffer will be modified and the size of the modified buffer will be returned.
- Assumptions:
  - The data_ptr will point to an array of bytes.
  - Each byte will contain a number from 0 to 127 (0x00 to 0x7F).
  - It is common for the data in the buffer to have the same value repeated in the series.
  - The compressed data will need to be decompressable.
  - Please ensure that your algorithm allows for a decompression algorithm to return the buffer to it's previous form.

    Example data and function call:

```
// Data before the call
data_ptr[] = { 0x03, 0x74, 0x04, 0x04, 0x04, 0x35, 0x35, 0x64,
0x64, 0x64, 0x64, 0x00, 0x00, 0x00, 0x00, 0x00,
0x56, 0x45, 0x56, 0x56, 0x56, 0x09, 0x09, 0x09 };
data_size = 24;
new_size = byte_compress( data_ptr, data_size );
```

1.2 Compression Algorithm pseudocode

1.2.1 General description

Inspired by the L4Z algorithm I found during my research, I chose this since it balances compression speed with compression size in order to ensure throughput. Ensuring throughput would be essential for applications like BLE data transmission on an embedded device.

I decided to focus on creating pairs of byte counts of sequences of duplicated bytes with a sample of the byte. These pairs would be inserted periodically within the data so that the data could be parsed in sequence or in portions if needeed. Parsing the data in portions would be practical in a transmission application where the receiving device could start decompressing the data without waiting for the transmission to complete.

1.2.2 Compression of matched sequences

I formatted the sequence byte counts into a half-byte (nibble) in order to minimize the size of the byte counts. I then put 2x nibbles together to form a "byte count token" which indicates the byte count of the preceding and following byte sequences. This helps parsing by byte-by-byte for simplicity. Finally only one example byte of each of the preceding and following byte sequences needs to be kept after compression, resulting in a reduction in overall byte count.

I decided to limit to using only the first 3x bits of the nibble, limiting the match recognition lengh to 7. This simplification allowed for an unused bit which I used in the un-matched sequence enhancement described below. This also reduced the complexity of test cases for this implementation.

Example data and compression result:

```
original bytes:
0x04, 0x04, 0x04, 0x35, 0x35
preceding byte count:  3
following byte count:  2
token (Hex):  0x32
after insertion into sequence:
0x04, 0x04, 0x04, 0x32, 0x35, 0x35
after removing un-necesarry sequence bytes:
0x04, 0x32, 0x35
```

1.2.3 Compression of un-matched sequences

During testing I found poor performance when dealing with multiple un-duplicated bytes in a row. A refinement I made is that, un-duplicated bytes would get treated as a single sequence of un-duplicated bytes putting a token at the start and end of the sequence. This reduced the amount of tokens inserted when there was no compression via byte sequence duplication.
If the input array starts with an unmatched sequence, we must input a token (0x8#) at the beginning to allow for our decompression strategy, see Decompression Algorithm pseudocode
I utilized the unused 0x8 nibble bit to indicate an unmatched sequence, the start nibble may optionally include the count of unmatched bytes but is not necesarry by design since this allows for fewer tokens for unmatched sequences longer than 7.
Improvement: for sequences longer than 7, add a length byte after the token

Basic Example data and un-duplication enhanced compression result:

```
pre compression size:  8 original bytes:
0x02, 0x02, 0x04, 0x05, 0x07, 0x03, 0x03, 0x03
preceding matched byte count:  2
following unmatched byte count:  3
following matched byte count:  3
before unmatched token (Hex):  0x2B
after unmatched token (Hex):  0x83
after insertion into sequence:
0x02, 0x02, 0x2B, 0x04, 0x05, 0x07, 0x83, 0x03, 0x03, 0x03
after removing un-necesarry sequence bytes:
0x02, 0x2B, 0x04, 0x05, 0x07, 0x83, 0x03
post compression size:  7
```

1.2.4 Overflow Handling

Since adding tokens to unmatched byte sequences results in a net gain in bytes, we need to add overlow capability to our algorithm to preserve later data in our input array. This overflow capability will unfortunately be cyclically costly as we have to create a gap in the data to insert the tokens.
Improvement: utilize an output memory space, rather than overwritting the input buffer, this could be an input to the function
Improvement: dynamically allocate more memory to the array, but this is typically disabled in embedded applications.

Overflow Example data and un-duplication enhanced compression result:

```
pre compression size:  3 original bytes:
0x04, 0x05, 0x07
preceding unmatched byte count:  3
following byte count:  3
before unmatched token (Hex):  0x8B
after unmatched token (Hex):  0x80
after insertion into sequence:
0x8B, 0x04, 0x05, 0x07, 0x80
after removing un-necesarry sequence bytes:
0x8B, 0x04, 0x05, 0x07, 0x83, 0x35
post compression size:  6 Results in more bytes than we started with!!!
```

1.3 Decompression Algorithm pseudocode

Similar to compression, decompression can be done in sequence and in segments.

- The first token will always be either in the first or second byte of the compressed array.
  - If the first byte contains a value larger than 0x7F then the file starts with an unmatched string.
  - If not, then the second byte is your first token. After Identifying the first token, you follow the compression rules in reverse to decompress the bytes.

- For a token nibble N in the compressed array, indicating a matched string, duplicate the next byte N times into the decompressed array. Then skip the following byte in the compressed array to find your next token.

- For a token nibble N in the compressed array, indicating an unmatched string, scan the following bytes for a value greater than 0x7F to identify the next tokena and the end of the unmatched length.
  Improvement: recommend that the output not be the same array as the compressed version for cyclic efficiency.
  Improvement: recommend that there be a header/footer showing how large the decompressed size is for usage for memory allocation

Example matched token data and decompression result:

```
original bytes:
0x04, 0x32, 0x35, 0x04, 0x32, 0x35
token1 (Hex):  0x32
preceding byte count:  3
following byte count:  2
token1 bytes decompressed:  0x04,0x04,0x04, 0x35, 0x35, token2 (Hex):  0x32
token2+Token1 bytes decompressed:  0x04,0x04,0x04, 0x35, 0x35, 0x04,0x04,0x04,
```

```
0x35, 0x35,
```

Example unmatched token data and decompression result:

```
pre compression size:  3 original bytes:
0x8B, 0x04, 0x05, 0x07, 0x83, 0x35
token1 (Hex):  0x8B
token2 (Hex):  0x83
preceding byte count:  0 unmatched sequence byte count:  3
following matched sequence byte count:  3
bytes decompressed:  0x04, 0x05, 0x07, 0x35, 0x35, 0x35
post compression size:  6
```

2 Unit Tests

2.1 Debug testing

During development, additional printf's were inserted into the algorithm to show step-by-step execution and results. I also used breakpoints to step through the code and observe the results.

```
Initialization complete
Size before:  24
{ 0x3, 0x74, 0x4, 0x4, 0x4, 0x35, 0x35, 0x64, 0x64, 0x64, 0x64, 0x0, 0x0,
0x0, 0x0, 0x0, 0x56, 0x45, 0x56, 0x56, 0x56, 0x9, 0x9, 0x9, }
//debug step-by-step
{ 0x8A, 0x3, 0x74, 0x4, 0x4, 0x4, 0x35, 0x35, 0x64, 0x64, 0x64, 0x64, 0x0,
0x0, 0x0, 0x0, 0x0, 0x56, 0x45, 0x56, 0x56, 0x56, 0x9, 0x9, } { 0x8A, 0x3,
0x74, 0x83, 0x4, 0x4, 0x35, 0x35, 0x64, 0x64, 0x64, 0x64, 0x0, 0x0, 0x0,
0x0, 0x0, 0x56, 0x45, 0x56, 0x56, 0x56, 0x9, 0x9, } { 0x8A, 0x3, 0x74, 0x83,
0x4, 0x35, 0x24, 0x64, 0x64, 0x64, 0x64, 0x64, 0x0, 0x0, 0x0, 0x0, 0x0,
0x56, 0x45, 0x56, 0x56, 0x56, 0x9, 0x9, } { 0x8A, 0x3, 0x74, 0x83, 0x4,
0x35, 0x24, 0x64, 0x0, 0x5A, 0x56, 0x45, 0x64, 0x0, 0x0, 0x0, 0x0, 0x56,
0x45, 0x56, 0x56, 0x56, 0x9, 0x9, } { 0x8A, 0x3, 0x74, 0x83, 0x4, 0x35,
0x24, 0x64, 0x0, 0x5A, 0x56, 0x45, 0x83, 0x56, 0x56, 0x9, 0x9, 0x9, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, }
Size Compressed:  16
Compress Time Taken:  7ms
{ 0x8A, 0x3, 0x74, 0x83, 0x4, 0x35, 0x24, 0x64, 0x0, 0x5A, 0x56, 0x45, 0x83,
0x56, 0x9, 0x30, }
Size decompressed:  0
Decompress Time Taken:  0ms
{ 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, }
```

2.2 Pass Fail Testing

Test results for pass/fail testing, testing against the sample input

```
Initialization complete
Size before:  24
{ 0x3, 0x74, 0x4, 0x4, 0x4, 0x35, 0x35, 0x64, 0x64, 0x64, 0x64, 0x0, 0x0,
0x0, 0x0, 0x0, 0x56, 0x45, 0x56, 0x56, 0x56, 0x9, 0x9, 0x9, }
Size Compressed:  16
```

```
Compress Time Taken:  0ms
{ 0x8A, 0x3, 0x74, 0x83, 0x4, 0x35, 0x24, 0x64, 0x0, 0x5A, 0x56, 0x45, 0x83,
0x56, 0x9, 0x30, }
Size decompressed:  0
Decompress Time Taken:  0ms
{ 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, }
```

2.3 Corner Case Testing

Test results for corner case testing, testing against chosen inputs which progressively test the algorithm more and more. Limited to what I could imagine would be difficult for the algorithm and do manually within the given time.

2.3.1 all sequences of 2, input size 24

2.3.2 half matched, then half unmatched sequences, input size 24

2.3.3 half unmatched, then half matched sequences, input size 24

2.3.4 all unmatched sequences, input size 24

2.3.5 sample input x2, input size 48

2.3.6 sample input +1, input size 25

2.4 Random/all Case Testing

Test results for random/all case testing. Typically you'd want to use an automated method to go through all possible input/output combinations when possible. Though outside of work I haven't setup a unit testing framework yet, like parasoft or polyspace unit test, which would help with this exact task.
I used chatgpt to generate the random arrays, which helped them be random

3 Data Structure Index

3.1 Data Structures

Here are the data structures with brief descriptions:

4 File Index

4.1 File List

Here is a list of all files with brief descriptions:

5 Data Structure Documentation

5.1 cmprss_token_t Union Reference

Data Fields

- uint8_t byte
- struct {
  uint8_t after: 4
  uint8_t before: 4
  };

5.1.1 Detailed Description

Definition at line 62 of file compression_test.c.

5.1.2 Field Documentation

[struct]

```
struct { ... }
```

after

```
uint8_t after
```

Definition at line 67 of file compression_test.c.

before

```
uint8_t before
```

Definition at line 68 of file compression_test.c.

byte

```
uint8_t byte
```

Definition at line 64 of file compression_test.c.

The documentation for this union was generated from the following file:

- C:/_GIT_REPOS/Example_C_repo/compression_test.c

6 File Documentation

6.1 C:/_GIT_REPOS/Example_C_repo/compression_test.c File Reference

```
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include <time.h>
#include <assert.h>
```

Data Structures

- union cmprss_token_t

Macros

- #define INPUT_SIZE 24
- #define BUFFER_SIZE 64
- #define TOKEN_INIT 0
- #define NIBBLE_MAX 0xF
- #define NIBBLE_NON_MATCH_BIT 0x8
- #define NIBBLE_VALUE_MASK 0x7
- #define PRINT_ROW_SIZE 8
- #define ERASED_BYTE 0xFF
- #define DEBUG 1

Typedefs

- typedef uint8_t buffer_element_t
- typedef uint64_t array_size_t

Functions

- void print_array (uint8_t *data_ptr, array_size_t data_size)

  prints the input array to the console in a formatted fashion
- cmprss_token_t getMatchLen (buffer_element_t *data_ptr, array_size_t i)
- int byte_compress (buffer_element_t *data_ptr, array_size_t data_size)

  compresses a byte array of data using a custom algorithm
- int byte_decompress (buffer_element_t *uncmprss_data_ptr, array_size_t uncmprss_data_size, buffer_element_t *cmprss_data_ptr, array_size_t cmpress_data_size)
- void main (void)

  calls compress and decompress on a test sample of data and times it

6.1.1 Macro Definition Documentation

BUFFER_SIZE

```
#define BUFFER_SIZE 64
```

Definition at line 49 of file compression_test.c.

DEBUG

```
#define DEBUG 1
```

Definition at line 144 of file compression_test.c.

ERASED_BYTE

```
#define ERASED_BYTE 0xFF
```

Definition at line 55 of file compression_test.c.

INPUT_SIZE

```
#define INPUT_SIZE 24
```

Definition at line 48 of file compression_test.c.

NIBBLE_MAX

```
#define NIBBLE_MAX 0xF
```

Definition at line 51 of file compression_test.c.

NIBBLE_NON_MATCH_BIT

```
#define NIBBLE_NON_MATCH_BIT 0x8
```

Definition at line 52 of file compression_test.c.

NIBBLE_VALUE_MASK

```
#define NIBBLE_VALUE_MASK 0x7
```

Definition at line 53 of file compression_test.c.

PRINT_ROW_SIZE

```
#define PRINT_ROW_SIZE 8
```

Definition at line 54 of file compression_test.c.

TOKEN_INIT

```
#define TOKEN_INIT 0
```

Definition at line 50 of file compression_test.c.

6.1.2 Typedef Documentation

array_size_t

```
typedef uint64_t array_size_t
```

Definition at line 60 of file compression_test.c.

buffer_element_t

```
typedef uint8_t buffer_element_t
```

Definition at line 59 of file compression_test.c.

6.1.3 Function Documentation

byte_compress()

```
int byte_compress (
            buffer_element_t * data_ptr,
            array_size_t data_size)
```

compresses a byte array of data using a custom algorithm

Parameters

data_ptr	
data_size	

Returns

int

Definition at line 152 of file compression_test.c.

byte_decompress()

```
int byte_decompress (
            buffer_element_t * uncmprss_data_ptr,
            array_size_t uncmprss_data_size,
            buffer_element_t * cmprss_data_ptr,
            array_size_t cmpress_data_size)
```

Parameters

data_ptr	
data_size	

Returns

int

Definition at line 326 of file compression_test.c.

getMatchLen()

```
cmprss_token_t getMatchLen (
            buffer_element_t * data_ptr,
            array_size_t i)
```

Definition at line 97 of file compression_test.c.

main()

```
void main (
            void )
```

calls compress and decompress on a test sample of data and times it

Definition at line 340 of file compression_test.c.

print_array()

```
void print_array (
            uint8_t * data_ptr,
            array_size_t data_size)
```

prints the input array to the console in a formatted fashion

Parameters

data_ptr	
data_size	

Definition at line 78 of file compression_test.c.

6.2 C:/_GIT_REPOS/Example_C_repo/compression_test.c

Go to the documentation of this file.

```
00001
00039 #include <stdio.h>
00040 #include <string.h>
00041 #include <stdint.h>
00042 #include <time.h>
00043
00044 #include <assert.h>
00045
00046
00047
00048 #define INPUT_SIZE 24
00049 #define BUFFER_SIZE 64
00050 #define TOKEN_INIT 0
00051 #define NIBBLE_MAX 0xF
00052 #define NIBBLE_NON_MATCH_BIT 0x8
00053 #define NIBBLE_VALUE_MASK 0x7
00054 #define PRINT_ROW_SIZE 8
00055 #define ERASED_BYTE 0xFF
00056
00057
00058
00059 typedef uint8_t buffer_element_t;
00060 typedef uint64_t array_size_t;
00061
```

```
00062 typedef union
00063 {
00064   uint8_t byte;
00065   struct
00066   {
00067     uint8_t after : 4;
00068     uint8_t before : 4;
00069   };
00070 } cmprss_token_t;
00071
00078 void print_array(uint8_t *data_ptr, array_size_t data_size)
00079 {
00080   printf("{");
00081   for (array_size_t k = 0; k < data_size; k++)
00082   {
00083     // start a new row for every PRINT_ROW_SIZE bytes
00084     if ((k % PRINT_ROW_SIZE) == 0)
00085     {
00086       printf("\n 0x%X, ", data_ptr[k]);
00087     }
00088     else
00089     {
00090       printf("0x%X, ", data_ptr[k]);
00091     }
00092   }
00093
00094   printf("\n}\n");
00095 }
00096
00097 cmprss_token_t getMatchLen(buffer_element_t *data_ptr, array_size_t i)
00098 {
00099   cmprss_token_t token1;
00100   token1.before = 0;
00101   token1.after = 0;
00102
00103   if (data_ptr[i] != ERASED_BYTE)
00104   {
00105     if (data_ptr[i] == data_ptr[i + 1])
00106     {
00107
00108       // original match
00109       token1.after++;
00110       // find the number of consecutive matches after the currVal
00111       for (uint8_t j = 1; j < NIBBLE_NON_MATCH_BIT; j++)
00112       {
00113
00114         if (data_ptr[i] == data_ptr[i + j])
00115           token1.after++;
00116         else
00117         {
00118           break;
00119         }
00120       }
00121     }
00122     else
00123     {
00124       // we should skip the first match, so we start at 2, since we don't what to include the final
      value in the case of a non-match length
00125       // find the number of consecutive non-matches after the currVal
00126       for (uint8_t j = 1; j < NIBBLE_NON_MATCH_BIT; j++)
00127       {
00128         if (data_ptr[i] == ERASED_BYTE)
00129           break;
00130         if (data_ptr[i + (j - 1)] == data_ptr[i + j])
00131           break;
00132         else
00133         {
00134           token1.after++;
00135           token1.after = token1.after | NIBBLE_NON_MATCH_BIT;
00136         }
00137       }
00138     }
00139   }
00140
00141   return token1;
00142 }
00143
00144 #define DEBUG 1
00152 int byte_compress(buffer_element_t *data_ptr, array_size_t data_size)
00153 {
00154   array_size_t size_after_compression = 0;
00155   array_size_t i = 0;
00156   uint8_t writeIndex = 0, eofWriteIndex = BUFFER_SIZE-1;
00157   cmprss_token_t token1, prevToken;
00158   token1.before = NIBBLE_MAX;
00159   token1.after = NIBBLE_MAX;
00160   prevToken.before = NIBBLE_MAX;
```

```
00161   prevToken.after = NIBBLE_MAX;
00162   buffer_element_t buffer = ERASED_BYTE, buffer2 = ERASED_BYTE;
00163   buffer_element_t eofBuffer[BUFFER_SIZE];
00164   memset(eofBuffer, ERASED_BYTE, BUFFER_SIZE);
00165   do
00166   {
00167     // get 2x consecutive sets of match/non-match sequences and set them as the lengths in the token
      along with their match bits
00168     token1.before = getMatchLen(data_ptr, i).after;
00169     token1.after = getMatchLen(data_ptr, i + (token1.before & NIBBLE_VALUE_MASK)).after;
00170
00171     if ((token1.before == 0) && (token1.after == 0))
00172       break;
00173
00174     //check if we're about to reach the end of the buffer
00175     if (data_size < (i+(token1.before & NIBBLE_VALUE_MASK)+1+(token1.after & NIBBLE_VALUE_MASK)))
00176     {
00177       if (data_ptr[i] == ERASED_BYTE)
00178         break;
00179       else if (eofWriteIndex+1 < BUFFER_SIZE)
00180       {
00181         //eofBuffer[eofWriteIndex--] = data_ptr[data_size-1];
00182         memmove(&data_ptr[writeIndex], &data_ptr[i], ((data_size)-(i)));
00183
00184         #ifdef DEBUG
00185         print_array(data_ptr, data_size);
00186         #endif
00187         memmove(&data_ptr[writeIndex+((data_size)-(i))], &eofBuffer[eofWriteIndex+1], BUFFER_SIZE -
      (eofWriteIndex+1));
00188
00189         #ifdef DEBUG
00190         print_array(data_ptr, data_size);
00191         #endif
00192         memset(&data_ptr[writeIndex+(BUFFER_SIZE - (eofWriteIndex+1))+((data_size)-(i))], 0xFF,
      ((data_size)-(writeIndex+(BUFFER_SIZE - (eofWriteIndex+1))+((data_size)-(i)))));
00193         eofWriteIndex = BUFFER_SIZE;
00194         i = writeIndex;
00195         #ifdef DEBUG
00196         print_array(data_ptr, data_size);
00197         #endif
00198
00199         continue;
00200       }
00201     }
00202
00203     if ((prevToken.after & NIBBLE_NON_MATCH_BIT) != 0)
00204     {
00205       //if we start off with a non matched streak, we need to put in a token at the beginning of the
      file
00206       if ((i != 0) && ((token1.before & NIBBLE_NON_MATCH_BIT) == 0) ||
00207           (i == 0) && ((token1.before & NIBBLE_NON_MATCH_BIT) != 0))
00208       {
00209         //add a token in to indicate the end of the non-matching characters
00210         token1.after = token1.before;
00211         token1.before = NIBBLE_NON_MATCH_BIT;
00212         // save the value from the space to be used by the token
00213         buffer = data_ptr[writeIndex];
00214       }
00215       else if (i==0)
00216       {
00217         //matching bit at the beginning of the array, move on
00218       }
00219       else
00220       {
00221         //we have non-matching characters which need to be continued
00222         memmove(&data_ptr[writeIndex], &data_ptr[i], (token1.before & NIBBLE_VALUE_MASK));
00223         writeIndex = writeIndex + (token1.before & NIBBLE_VALUE_MASK);
00224         i = i + (token1.before & NIBBLE_VALUE_MASK);
00225         continue;
00226       }
00227     }
00228
00229     // WARNING inserting a token will increase the size not reduce it
00230     if (((token1.before & NIBBLE_NON_MATCH_BIT) != 0) &&
00231         ((token1.after & NIBBLE_NON_MATCH_BIT) != 0) &&
00232         (writeIndex > (i - 1)))
00233     {
00234       // If we quit here, this failure mode will have corrupted the data due to the incomplete
      conversion
00235       // TODO create recovery method? or way to continue compressing? Perhaps make a buffer and
      shuffle all remaining bits outwards...
00236       size_after_compression = (array_size_t)-1;
00237       break;
00238     }
00239
00240     if ((token1.before & NIBBLE_NON_MATCH_BIT) != 0)
00241     {
```

```
00242       memmove(&data_ptr[writeIndex], &data_ptr[i], (token1.before & NIBBLE_VALUE_MASK));
00243       writeIndex = writeIndex + (token1.before & NIBBLE_VALUE_MASK);
00244       // save the value from the space to be used by the token
00245       buffer = data_ptr[writeIndex];
00246     }
00247     else
00248     {
00249       // insert the before match value
00250       data_ptr[writeIndex] = data_ptr[i];
00251       writeIndex = writeIndex + 1;
00252 
00253       // save the value from the space to be used by the token
00254       buffer = data_ptr[writeIndex];
00255     }
00256     #ifdef DEBUG
00257     print_array(data_ptr, data_size);
00258     #endif
00259     i = i + (token1.before & NIBBLE_VALUE_MASK);
00260 
00261     // insert the token
00262     data_ptr[writeIndex++] = (buffer_element_t)token1.byte;
00263 
00264     #ifdef DEBUG
00265     print_array(data_ptr, data_size);
00266     #endif
00267 
00268     if (token1.after == 0)
00269       break;
00270 
00271     if ((token1.after & NIBBLE_NON_MATCH_BIT) != 0)
00272     {
00273       if (buffer != ERASED_BYTE)
00274       {
00275         if ((writeIndex + (token1.after & NIBBLE_VALUE_MASK)) < i)
00276         {
00277           memmove(&data_ptr[writeIndex+1], &data_ptr[writeIndex], (token1.after & NIBBLE_VALUE_MASK));
00278           data_ptr[writeIndex] = buffer;
00279           #ifdef DEBUG
00280           print_array(data_ptr, data_size);
00281           #endif
00282         }
00283         else
00284         {
00285           //problem case where we've not got enough space to insert the token
00286           eofBuffer[eofWriteIndex--] = data_ptr[data_size-1];
00287           memmove(&data_ptr[writeIndex+1], &data_ptr[writeIndex], ((data_size-1)-(writeIndex)));
00288           data_ptr[writeIndex] = buffer;
00289           #ifdef DEBUG
00290           print_array(data_ptr, data_size);
00291           #endif
00292           i = i + 1;
00293         }
00294       }
00295 
00296       memmove(&data_ptr[writeIndex], &data_ptr[i], (token1.after & NIBBLE_VALUE_MASK));
00297       writeIndex = writeIndex + (token1.after & NIBBLE_VALUE_MASK);
00298     }
00299     else
00300     {
00301       // insert the before match value
00302       data_ptr[writeIndex] = data_ptr[i+1];
00303       writeIndex = writeIndex + 1;
00304     }
00305     #ifdef DEBUG
00306     print_array(data_ptr, data_size);
00307     #endif
00308 
00309     i = i + (token1.after & NIBBLE_VALUE_MASK);
00310     buffer = ERASED_BYTE;
00311     prevToken.byte = token1.byte;
00312   } while ((i < data_size) || (data_ptr[i] != ERASED_BYTE));
00313 
00314   size_after_compression = writeIndex;
00315 
00316   return size_after_compression;
00317 }
00318 
00326 int byte_decompress(buffer_element_t *uncmprss_data_ptr, array_size_t uncmprss_data_size,
      buffer_element_t *cmprss_data_ptr, array_size_t cmpress_data_size)
00327 {
00328   array_size_t size_after_compression = 0;
00329   array_size_t i = 0;
00330 
00331 
00332 
00333   return size_after_compression;
00334 }
```

```
00335
00340 void main(void)
00341 {
00342   clock_t start_time, end_time;
00343   uint64_t time_taken = 0.0f;
00344   /*
00345   uint8_t input_data_ptr[INPUT_SIZE] = {0x03, 0x74, 0x04, 0x04, 0x04, 0x35, 0x35, 0x64,
00346                                         0x64, 0x64, 0x64, 0x00, 0x00, 0x00, 0x00, 0x00,
00347                                         0x56, 0x45, 0x56, 0x56, 0x56, 0x09, 0x09, 0x09};
00348   */
00349
00350   uint8_t input_data_ptr[INPUT_SIZE] = {
00351     0xAA, 0xAA, 0xBB, 0xBB, 0xCC, 0xCC, 0xDD, 0xDD,
00352     0xEE, 0xEE, 0x11, 0x11, 0x22, 0x22, 0x33, 0x33,
00353     0x44, 0x44, 0x66, 0x77, 0x88, 0x99, 0x00, 0x00
00354 };
00355 //  correct compression
00356 /*
00357     0xAA, 0x22, 0xBB, 0xCC, 0x22, 0xDD, 0xEE, 0x22,
00358     0x11, 0x22, 0x22, 0x33, 0x44, 0x2C, 0x66, 0x77,
00359     0x88, 0x99, 0x82, 0x00
00360 };
00361 */
00362   // TODO  instead of having the decompressed size known ahead of time, we should use malloc or
      similar since all we would know is the compressed size
00363   uint8_t decompressed_data_ptr[INPUT_SIZE] = {0};
00364   uint64_t main_data_size = INPUT_SIZE;
00365   uint64_t main_cmprss_size = main_data_size;
00366   uint64_t main_decmprss_size = main_data_size;
00367
00368   printf("\n\nInitialization complete\n");
00369   printf("Size before: %d\n", main_data_size);
00370   print_array(input_data_ptr, main_data_size);
00371
00372
00373   start_time = clock();
00374   main_cmprss_size = byte_compress(input_data_ptr, main_data_size);
00375   end_time = clock();
00376
00377   // TODO write 0xFF to all bytes larger than the post-compression size?
00378
00379   // check that clock ticks are indeed seconds
00380   assert(CLOCKS_PER_SEC == 1000);
00381   // Calculate the time difference in milliseconds
00382   time_taken = (uint64_t)(end_time - start_time);
00383
00384   printf("\nSize Compressed: %d\n", main_cmprss_size);
00385   printf("Compress Time Taken: %dms\n", time_taken);
00386   print_array(input_data_ptr, main_cmprss_size);
00387
00388   start_time = clock();
00389   main_decmprss_size = byte_decompress(input_data_ptr, main_data_size, decompressed_data_ptr,
      main_cmprss_size);
00390   end_time = clock();
00391
00392   // Calculate the time difference in milliseconds
00393   time_taken = (uint64_t)(end_time - start_time);
00394
00395   printf("\nSize decompressed: %d\n", main_decmprss_size);
00396   printf("Decompress Time Taken: %dms\n", time_taken);
00397   print_array(decompressed_data_ptr, main_data_size);
00398
00399   // TODO print compressed array
00400
00401   return;
00402 }
```

6.3 doc_pages/main_page.md File Reference

6.4 doc_pages/testing_page.md File Reference

Index