roadC

Generated by Doxygen 1.9.1

Chapter 1

roadC

Author

Dr. Steffen Görzig

Version

1.0

Date

2012-2021

Copyright

Standard data compression techniques cannot be applied for traditional embedded systems since these systems are typically too restricted by timing constraints or by memory limitations. Read-Only Array Data Compaction (roadC) addresses this problem: it is an approach for compression of read-only array data. The main advantage over standard approaches is the ability to access compressed data without any decompression, i.e. without any extra computational and memory costs.

The roadC library is a reference implementation of the roadC algorithm. It is implemented in C and can be compiled using a C or C++ compiler. This manual contains:

- Introduction to the roadC algorithm
- Implementation details of the roadC reference implementation
- Reference manuel for the roadC reference implementation functions
- Examples how to apply the roadC reference implementation

Chapter 2

Introduction

Data compression has been a well-known approach for many years and is successfully applied for various use cases, e.g. to reduce data storage space or to increase data transmission capacity. However, all current compression techniques share the same drawback: to access compressed data the data must be decompressed, and decompression imposes extra computational and memory costs. For personal computers or smartphones, these costs are usually acceptable for many use cases. But a wide range of embedded systems cannot use data compression at all due to

- Timing constraints: for example, when certain startup or response times have to be ensured.
- Memory limitations: the additional memory consumption for the decompressed data as well as for the decompression algorithm itself can be too large when only a few kilobytes of RAM/ROM memory are available.

Read-Only Array Data Compaction (roadC) addresses this problem.

2.1 Read-Only Array Data Compaction (roadC)

Read-Only Array Data Compaction is a new approach for lossless and lossy compression. In contrast to other compression algorithms, roadC allows avoiding additional computation time or memory consumption when accessing the data, i.e. no decompression at all. The roadC algorithm consists of three steps:

1. Input transformation
2. Data compaction
3. Output generation

2.1.1 Step 1: Input Transformation

Note

This step is not part of the roadC reference implementation!

The input data is read-only arrays. The arrays can be of any length, dimension and type - as long as they are finite. To prepare the data for the next step, all input data has to be transformed to a set of one-dimensional arrays of the basic type byte. The input transformation algorithm in pseudo-code:

```
transformedArraysList = {}
while (inputReadOnlyArraysList!={}):
  inputArray = inputReadOnlyArraysList.pop()
  oneDimensionalArray = getNextOneDimensionalArray(inputArray)
  while (oneDimensionalArray!={}):
    transformedArray = transformToBasicType(oneDimensionalArray)
    transformedArraysList.append(transformedArray)
    oneDimensionalArray = getNextOneDimensionalArray(inputArray)
```

The called functions are:

- `pop()` Returns and removes the last item in the list.
- `append()` Appends a given array to the end of a list.
- `getNextOneDimensionalArray(array)` Decomposes a multidimensional input array into its one dimensional parts: returns the one dimensional arrays one by one at each call of the function. If the input array is one dimensional, return the array itself. Return an empty array when the last one dimensional array was already returned before.
- `transformToBasicType(array)` Returns a given one dimensional array transformed into a one dimensional array of basic type “byte”.

The transformation to arrays of the basic type byte depends on the platform representation of data types. Typically the following aspects have to be considered:

- Number of bytes for a specific type (e.g. two or four bytes for the representation of type “integer”).
- Endianness (e.g. little-endian or big-endian).
- Binary representation of negative values (e.g. two’s complement).

2.1.2 Step 2: Data Compaction

Note

This step is the core functionality of roadC and it is provided by the roadC reference implementation!

The input for the data compaction step is the set of one-dimensional byte arrays provided by the input transformation step. The core of the roadC algorithm is to reduce this array set. The result could again be a set of one-dimensional byte arrays; for practical reasons this set is reduced to only a single array.

So the roadC problem is: Build a single one-dimensional byte array of minimum size containing all input data.

The shortest superstring problem is a subset of the given roadC problem: Given a set of strings S={s1,s2. . . ,sn} over a finite alphabet T, a superstring of S is a string that contains each s(i) as a contiguous substring. The shortest superstring problem is to find a superstring of minimum length.

Since this problem is NP-Complete, approximation algorithms are applied: remove sub-arrays and the greedy approach.

2.1.2.1 Remove Sub-Arrays

If an array is completely part of another array, it is removed from the list. Example of the algorithm using byte arrays as transformed in step 1:

```
{16, 32}
{0, 16, 32, 128}
{1, 17}
{1, 17}
```

Resulting arrays after removing sub-arrays:

```
{0, 16, 32, 128}
{1, 17}
```

2.1.2.2 Greedy Approach

The greedy approach is to determine the two most overlapping arrays (left or right) in the list and replace them with the array obtained by overlapping them. Repeat this until only one array is left. Example of the algorithm using transformed byte arrays:

```
{0, 16, 155}
{155, 17, 0, 16}
{155, 233, 0}
```

Calculate overlapping data (left and right) for each combination:

```
{0, 16, 155}     and {155, 17, 0, 16} : overlap {155}
{155, 17, 0, 16} and {0, 16, 155}     : overlap {0, 16} -> largest overlap!
{0, 16, 155}     and {155, 233, 0}    : overlap {155}
{155, 233, 0}    and {0, 16, 155}     : overlap {0}
{155, 17, 0, 16} and {155, 233, 0}    : overlap {}
{155, 233, 0}    and {155, 17, 0, 16} : overlap {}
Concatenate arrays with largest overlap {155, 17, 0, 16} and {0, 16, 155} to {155, 17, 0, 16, 155}
Remove {155, 17, 0, 16} and {0, 16, 155} from input list
Add {155, 17, 0, 16, 155} to list
Resulting byte arrays after one greedy approach step:
{155, 17, 0, 16, 155}
{155, 233, 0}
Resulting byte array after two greedy approach steps:
{155, 17, 0, 16, 155, 233, 0}
```

Different strategies can be applied to choose two arrays when the best overlapping data is of the same size for more than one combination. For example, a strategy can be to take the first combination, the last, or a random one. This will of course influence the result, but practical results show the difference is negligibly small. For speedup, the greedy approach should avoid unnecessary overlap calculations. When there is no overlap anymore the remaining arrays are then concatenated to a single array. Although not proven, a straightforward greedy approach is conjectured to give an approximation ratio of 2.

2.1.3 Step 3: Output Generation

Note

This step is not part of the roadC reference implementation!

The output generation step replaces the input arrays by pointers to the one-dimensional byte array provided by the compaction step. The output generation algorithm in pseudo-code:

```
outputArraysList = {}
while (inputReadOnlyArraysList!={}):
  inputArray = inputReadOnlyArraysList.pop()
  oneDimensionalArray = getNextOneDimensionalArray(inputArray)
  while (oneDimensionalArray!={}):
    transformedArray = transformToBasicType(oneDimensionalArray)
    pointer = compactedData.getPosition(transformedArray)
    replaceCurrentOneDimensionalArrayByPointer(inputArray, pointer)
    oneDimensionalArray = getNextOneDimensionalArray(inputArray)
  outputArraysList.append(inputArray)
generateTargetCode(outputArraysList)
```

The called functions are (on top of the described functions in Step 1: Input Transformation):

- `getPosition(array)` Returns the lowest index as a pointer where "array" is found as subarray in the compacted byte array.
- `replaceCurrentOneDimensionalArrayByPointer(array, pointer)` Replace a one dimensional sub-array in array by a given pointer. The one dimensional sub-array to be replaced is the array given by the last call of getNextOneDimensionalArray(array).
- `generateTargetCode(array)` Generate code for the target programming language.

The code generation strongly depends on the target programming language. However typically the following points have to be taken into account:

- Pointer conversion to the types of the input data.
- Additional array pointer variables for arrays with dimension > 2.

Warning

Please note that replacing array variables by pointers or adding pointer variables may lead to additional memory consumption depending on the size of the pointer representation. Therefore compacted data sizes can become larger than the original input data sizes if compaction has no or only little effect.

2.2 Memory Handling

Memory handling of CPUs can require special demands on the way data is stored in the memory. In this context roadC can handle data alignment, data structure padding, and memory protection.

2.2.1 Data Alignment

Data alignment restricts the start address of data. Typically the address is some multiple of the computer's word size. If no special alignment is needed, the alignment must be set to 1.

The result of the roadC algorithm is only one byte array, all input arrays are then part of this byte array. The alignment of this array is calculated from all alignments of the input arrays. By using this calculated alignment every single alignment of the input arrays is still valid, although they are now part of one large byte array.

Here is a simple example:
```
data1: {233, 10, 200}   alignment: 1
data2: {155, 17, 0, 16} alignment: 4
data3: {16, 233, 10}    alignment: 8
Result:
{16, 233, 10, 200, 155, 17, 0, 16} alignment: 8
position in resulting array indices: data1:1 data2:4 data3:0
```

2.2.2 Data Structure Padding

Using the roadC algorithm without taking care of data structure padding will produce valid results. But taking care of padding bytes or even padding bits can improve the compression rate. This is because roadC searches for matching data in arrays. Since the data content of padding bytes or padding bits can be ignored, the probability of matches increases.

The roadC algorithm handles data structure padding on bit level. For each data byte, there is an associated padding byte that determines which bit in the data byte contains information (padding bit: 0) an which is meaningless (padding bit: 1). E.g. a padding byte value of 1 indicated that bit 0 in the data byte is a padding bit and bits 1-7 contain information. A value of 128 indicates bit 7 in the data byte is a padding bit, a value of 129 indicates bit 1 and bit 7 are padding bits and so on. A value of 255 indicates that all bits are padding bits, i.e. the data byte is a padding byte. All padding information for an input data array is given by an array of the same size and is called padding byte mask array. If no padding byte mask array is given all data are considered to be meaningful i.e. no padding bytes/bits are used.

The following example shows a roadC calculation with and without the use of padding information:
```
data1:            {2, 3, 3, 4}
data1PaddingMask: {1, 2, 0, 0}
data2:            {1, 3, 3}
data2PaddingMask: {255, 255, 0}
data3:            {5, 7}
data3PaddingMask: {255, 252}
Result without padding information:
{2,3,3,4,1,3,3,5,7}
position in resulting array indices: data1:0 data2:4 data3:7
Result with padding information:
{2, 3, 3, 4} paddingMask: {1,0,0,0}
position in resulting array indices: data1:0 data2:0 data3:0
```

2.2.3 Memory Protection

The result of the roadC algorithm is one byte array and all input arrays are now part of this byte array. This could be a problem when using memory protection (e.g. a memory management unit): the single array is not accessible for all input data if the input data belong to different memory segments. To overcome this problem roadC must be applied separately for each segment, i.e. for each memory segment containing roadC input data the roadC algorithm is applied and for each of these calculations the input arrays must belong to the same memory segment.

```
Input arrays:
{233, 10, 200}   memory segment: 1
{155, 17, 0, 16} memory segment: 2
{16, 155, 17}    memory segment: 2
{200, 11}        memory segment: 1
Calculate memory segment 1:
{233, 10, 200}
{200, 11}
Result memory segment 1:
{233, 10, 200, 11}
Calculate memory segment 2:
{155, 17, 0, 16}
{16, 155, 17}
Result memory segment 2:
{16, 155, 17, 0, 16}
```

2.3 Example in C

The following example may give a better understanding of how roadC works. The Remove Sub-Arrays and Greedy approach are applied for each data compaction step until only one single byte array is left. The example is given in the C programming language to explain how roadC is used on a real platform. The input arrays are of different lengths, dimensions, and types: In the example, some array properties are encoded in the array identifier. The first identifier part indicates the type (`uchar` means unsigned byte, `schar` means signed byte, and `int` means integer). For a better readability the const-correctness (keyword `const`) is omitted for the read-only arrays:
```c
unsigned char ucharData[5] = {0,128,0,255,255};
```

```
int intData1[3] = {-32768,-256,-1};
int intData2[3] = {-1,32767,256};
signed char scharData1[2] = {-128,0};
signed char scharData2[2] = {0,-1};
signed char scharData3[2] = {0,1};
signed char scharData4[2] = {-1,127};
```

In the following a small platform is assumed using two's complement for negative values and a two-byte big-endian representation for integer. The first step, input transformation, transforms the one-dimensional parts of the arrays into arrays of the basic type byte according to the given assumptions:

```
unsigned char ucharData[5] = {0,128,0,255,255};
unsigned char intData1[6] = {0,128,0,255,255,255};
unsigned char intData2[6] = {255,255,255,127,0,1};
unsigned char scharData1[2] = {128,0};
unsigned char scharData2[2] = {0,255};
unsigned char scharData3[2] = {0,1};
unsigned char scharData4[2] = {255,127};
```

The resulting set of one-dimensional byte arrays is handed over to step 2, data compaction. During compaction first all sub-arrays are removed:

```
unsigned char intData1[6] = {0,128,0,255,255,255};
unsigned char intData2[6] = {255,255,255,127,0,1};
```

After that the greedy approach is applied, the result is the compacted byte array called `cba`:

```
unsigned char cba[9] = {0,128,0,255,255,255,127,0,1};
```

The last step is output generation replacing the input arrays by pointers to the compacted byte array:

```
unsigned char cba[9] = {0,128,0,255,255,255,127,0,1};
unsigned char *ucharData = (unsigned char*)&cba[0];
int intData1[3] = (int*)&cba[0];
int intData2[3] = (int*)&cba[3];
signed char scharData1[2] = (signed char*)&cba[1];
signed char scharData2[2] = (signed char*)&cba[2];
signed char scharData3[2] = (signed char*)&cba[7];
signed char scharData4[2] = (signed char*)&cba[5]};
```

In this example the number of bytes is reduced from 25 (input arrays) to 9 (compacted byte array). On the other hand, new pointers are introduced, which increases the overall data size.

2.3.1 Multidimensional Arrays

RoadC can also be applied for multidimensional arrays. For practical reasons this is not recommended, as described in section Restrictions. The next example will explain how multidimensional arrays can be handled. Consider the following input data:

```
unsigned char uchar1D[5] = {0,128,0,255,255};
int int2D[2][3] = {{-32768,-256,-1},{-1,32767,256}};
signed char schar3D[2][2][2] = {{{-128,0},{0,-1}},{{0,1},{-1,127}}};
```

Again using two's complement for negative values and a two-byte big-endian representation for integer is assumed. C code after the 3 steps of the roadC algorithm:

```
unsigned char cba[9] = {0,128,0,255,255,255,127,0,1};
unsigned char *uchar1D = (unsigned char*)&cba[0];
int *int2D[2] = {(int*)&cba[0],(int*)&cba[3]};
signed char *schar3D0[2] = {(signed char*)&cba[1],(signed char*)&cba[2]};
signed char *schar3D1[2] = {(signed char*)&cba[7],(signed char*)&cba[5]};
signed char **schar3D[2] = {schar3D0,schar3D1};
```

2.4 Restrictions

Read-only data compaction is a transformation which can cause restrictions when used with a real programming language like C. If roadC were part of a compiler, most of the points mentioned below could be handled by the compiler itself. Some points must be handled by the programmer or depend on the hardware used.

As shown with type "int", the input transformation must be aware of compiler- or platform-dependent types (e.g. for "enum", "wchar_t", "struct"), padding bytes, and for memory alignment. In the C example, arrays are replaced by

pointers. This is an indirection which can result in different machine code. Array element addresses can no longer be calculated by start address plus offset. Now, there is a read access to the pointer variable. Since this is slower on typical hardware, this can be considered as decompression cost.
Pointers make no difference for the array subscript operator “[]”. But they make a difference e.g. for the address operator “&”. This is because the declaration between arrays and pointers differs, which has to be taken into account by programmers (e.g. functions expecting arrays as arguments can no longer be used with pointers). Since the input arrays are replaced by pointers to the compacted data, data access must not be allowed that assumes a contiguous representation of multi-dimensional arrays in memory when using compacted data.

Uncompacted original:
```
unsigned int i;
unsigned char arr[2][2] = {{1, 2}, {2 ,4}};
/* next data access allowed for uncompacted code but NOT for compacted code: */
unsigned char *arrPtr = &arr[0][0];
for (i=0; i<2*2; i++){
  printf("%u ", (*arrPtr));
  ++arrPtr;
}
```

Compacted version:
```
unsigned int i, j;
/* compacted version: */
unsigned char cba[3] = {1, 2, 4};
unsigned char *arr[2] = {(unsigned char*)&cba[0], (unsigned char*)&cba[1]};
/* for compacted and uncompacted code allowed: */
for (i=0; i<2; i++){
  for (j=0; j<2; j++){
    printf("%u ", arr[i][j]);
  }
}
```

Due to the same reason, functions like `memcpy` should not be used to copy data from multi-dimensional compacted data to uncompacted data. Also it cannot be assumed any more that arrays have the original positions in memory, e.g. it is not allowed to access multiple arrays with a loop assuming contiguous positions of data in the memory. Functions like `sizeof` or `offsetof` should be handled with care. If data compaction is performed automatically (e.g. as part of a compiler), it must be ensured that this kind of data accesses does not occur. This can be achieved by data-flow analysis. Analysis itself is hindered by the need for pointer analysis instead of simpler array analysis. Another topic depends on the hardware: if the hardware uses bank switching for memory, the compacted data array might no longer be accessible in every context (see section Memory Protection how to solve this problem).

Chapter 3

Implementation Details

The basic functionality of roadC is described in the Introduction part. It is recommended to read this chapter first before continuing with the implementation details.

The core functionality of roadC is closely related the shortest common superstring problem (SCS): to find the shortest possible string that contains every string in a given set as substrings. Since this problem is NP-complete approximation algorithms are applied, e.g. the greedy approach. Although not proven, a straightforward greedy approach is conjectured to give an approximation ratio of 2.

The first step of roadC is to apply a very straight forward algorithm to reduce the input data array set: remove sub-arrays. The second step is to apply the greedy approach. Both steps have to take data alignment and data structure padding into account. Therefore these topics are explained first.

3.1 Data Alignment

Data alignment restricts the start address of a data array. Typically the address is some multiple of the computer's word size to improve the system's performance. If no special alignment is needed, the alignment must be set to 1. Taking data alignments into account can of course result in a lower compression rate.

The roadC algorithm has to ensure that the given data alignments of all input data are still valid in the resulting compressed array. To allow a fast calculation the input data itself is not modified at first, instead two values are used for each array:

```
unsigned int alignment;
unsigned int alignmentOffset;
```

The variable `alignment` is the current alignment of the array, the variable `alignmentOffset` is the number of padding bytes to be added before the array to ensure a certain alignment. When adding an input array the `alignment` is given as a parameter, `alignmentOffset` is set to `0`.

The roadC algorithm uses three different methods to handle alignment during calculation:

- remove sub-arrays,
- concatenate arrays with overlap, and
- concatenate arrays without overlap.

The first two methods may have no solution depending on given alignment values for the arrays. But there is always a solution for the third method. This ensures that alignment handling can be used for roadC in the first place, i.e. that there is always a solution for any given alignment inputs.

3.1.1 Data Alignment when Removing Sub-Arrays

When a sub-array is identified the data alignment calculation is applied. The results can be:

1. no solution
2. valid solution

In the first case the sub-array cannot be removed. The following example has no solution:
```
data1: {1, 2, 3, 4}   alignment: 1 alignmentOffset: 0
data2: {3, 4}         alignment: 4 alignmentOffset: 0
Result:
-
```

For the next example a solution can be calculated:
```
data1: {1, 2, 3, 4}   alignment: 4 alignmentOffset: 0
data2: {3, 4}         alignment: 2 alignmentOffset: 6
Result:
{1, 2, 3, 4} alignment: 4 alignmentOffset: 4
```

3.1.2 Data Alignment when concatenating arrays with overlap

When concatenating arrays with overlap during the greedy approach, three cases can occur:

1. no solution
2. solution leads to a very large array
3. valid solution

In the first two cases no concatenation with overlap is done. The following example has no solution:
```
data1: {1, 2, 3, 4}   alignment: 4 alignmentOffset: 0
data2: {4, 5}         alignment: 4 alignmentOffset: 0
Result:
-
```

For the next example a solution can be calculated, but the resulting array is very large. The limit to indicate that a resulting array is too large is given by the resulting array size when concatenating arrays without overlap. Since it is not clear that exactly these two arrays are concatenated at the end of the roadC algorithm this is of course only an approximated limit:
```
data1: {1, 2, 3, 4}   alignment: 3 alignmentOffset: 0
data2: {3, 4}         alignment: 5 alignmentOffset: 0
Result concatenation with overlap:
{1, 2, 3, 4, 5} alignment: 15 alignmentOffset: 12 size: 17
Result concatenation without overlap:
{1, 2, 3, 4, 0, 4, 5} alignment: 15 alignmentOffset: 0 size: 7
```

The next example has a valid solution and the arrays are concatenated:
```
data1: {1, 2, 3, 4}   alignment: 4 alignmentOffset: 0
data2: {4, 5}         alignment: 1 alignmentOffset: 0
Result concatenation with overlap:
{1, 2, 3, 4, 5} alignment: 4 alignmentOffset: 0 size: 5
Result concatenation without overlap:
{1, 2, 3, 4, 4, 5} alignment: 4 alignmentOffset: 0 size: 6
```

3.1.3 Data Alignment when concatenating arrays without overlap

It is always possible to calculate the data alignment when concatenating arrays without overlap. Thus this is the fallback solution for all cases which had no solution when removing sub-arrays or concatenating arrays with overlap.

When concatenating arrays without overlap, two cases can occur:

1. the solution is shorter in given array order, or
2. the solution is shorter in reverse order

In the next example the given array order produces the better solution:

```
data1: {1, 2, 3, 4}   alignment: 3 alignmentOffset: 0
data2: {5, 6}         alignment: 2 alignmentOffset: 0
Result in given order:
{1, 2, 3, 4, 5, 6} alignment: 6 alignmentOffset: 0 size: 6
Result in reverse order:
{5, 6, 0, 1, 2, 3, 4} alignment: 6 alignmentOffset: 0 size: 7
```

This example shows that the reverse order can also produce a shorter result:

```
data1: {1, 2, 3, 4}   alignment: 2 alignmentOffset: 0
data2: {5, 6}         alignment: 3 alignmentOffset: 2
Result in given order:
{1, 2, 3, 4, 0, 0, 0, 0, 5, 6} alignment: 6 alignmentOffset: 0 size: 10
Result in reverse order:
{5, 6, 1, 2, 3, 4} alignment: 6 alignmentOffset: 2 size: 8
```

3.2 Data Structure Padding

As mentioned in section Data Structure Padding roadC handles data structure padding on the finest granularity possible, on bit level. Thus even padding bits can lead to better compression results.

The roadC algorithm uses three different methods to handle data structure padding during calculation:

- remove sub-arrays,
- concatenate arrays with overlap,
- concatenate arrays without overlap, and
- Data Structure Padding when searching for an input array position in the resulting compressed array.

All methods use the standard logical operators negation (`not`), conjunction (`and`), and disjunction (`or`) a direct setting of values if needed.

3.2.1 Data Structure Padding when removing sub-arrays

When removing sub-arrays following data structure padding must be taken into account for:

1. the match calculation of sub-arrays
2. the adaption of the resulting array values and padding mask when a match was made

The next example shows these points:
```
data1:            {1, 3, 3}
data1PaddingMask: {2, 1, 255}
data2:            {3, 2}
data2PaddingMask: {6, 0}
Result:
{1, 2, 3} paddingMask: {2, 0 , 255}
position in resulting array indices: data1:0 data2:0
```

The matching is done with binary numbers, here for element at position 0:
```
data1[0]              00000001
data1PaddingMask[0]   00000010
relevantBits(data1[0]) 000000 1
data2[0]              00000011
data2PaddingMask[0]   00000110
relevantBits(data2[0]) 00000  1
```

In this case the relevant bits of data1 and data2 are equal, so there is match at position 0.

The same calculation for position 1:
```
data1[1]              00000011
data1PaddingMask[1]   00000001
relevantBits(data1[1]) 0000001
data2[1]              00000010
data2PaddingMask[1]   00000000
relevantBits(data2[1]) 00000010
```

The relevant bits matches again, therefore `data2` is a sub-array of `data1` and can be removed. The values and the padding mask of the remaining array must be adapted to the restriction given by both arrays. Otherwise it cannot be ensured that the removed array is still part of the resulting array after the roadC algorithm.

We start with the element at position 0:
```
data1[0]              00000001
data1PaddingMask[0]   00000010
relevantBits(data1[0]) 000000 1
data2[0]              00000011
data2PaddingMask[0]   00000110
relevantBits(data2[0]) 00000  1
resultData[0]         00000001
resultPaddingMask[0]  00000010
```

The resulting data keeps the values of both array data for the relevant bits. The padding mask is reduced to where both input padding mask bits are 1. That ensures that the relevant bits of both input arrays are still valid in the resulting padding mask. Same here for position 1:
```
data1[1]              00000011
data1PaddingMask[1]   00000001
relevantBits(data1[1]) 0000001
data2[1]              00000010
data2PaddingMask[1]   00000000
relevantBits(data2[1]) 00000010
resultData[1]         00000010
resultPaddingMask[1]  00000000
```

The data and padding mask at position 3 is not affected and therefore not modified.

3.2.2 Data Structure Padding when concatenating arrays with overlap

The handling of data structure padding when concatenating arrays with overlap is basically the same as when handling sub-arrays:

1. the match calculation of overlaps
2. the adaption of the resulting array values and padding mask when a match was made
3. new padding bytes are added due to alignment calculation

The next example shows the first two points:

```
data1:            {1, 2, 31}
data1PaddingMask: {2, 129, 28}
data2:            {3, 227, 4}
data2PaddingMask: {1, 224, 1}
Result:
{1, 2, 3, 4} paddingMask: {2, 1, 0 , 1}
position in resulting array indices: data1:0 data2:1
```

The non-overlapping parts are not modified, the overlapping part is calculated the same way as when removing sub-arrays. Here the calculation for the overlapping part:

```
data1[1]                00000010
data1PaddingMask[1]     10000001
relevantBits(data1[1])   000001
data2[0]                00000011
data2PaddingMask[0]     00000001
relevantBits(data2[0])  0000001
resultData[1]           00000010
resultPaddingMask[1]    00000001
data1[2]                00011111
data1PaddingMask[2]     00011100
relevantBits(data1[2])  000   11
data2[1]                11100011
data2PaddingMask[1]     11100000
relevantBits(data2[1])     00011
resultData[2]           00000011
resultPaddingMask[2]    00000000
```

New padding bytes can be added due data alignment. The padding bytes get the value `in` the array and value `255` in the padding byte mask to indicate every bit is a padding bit:

```
data1:            {1, 2, 31}
data1PaddingMask: {2, 129, 28}
data1Alignment:   1
data2:            {3, 227, 4}
data2PaddingMask: {1, 224, 1}
data2Alignment:   3
Result:
{0, 0, 1, 2, 3, 4} paddingMask: {255, 255, 2, 1, 0 , 1} alignment: 3
position in resulting array indices: data1:2 data2:3
```

3.2.3 Data Structure Padding when concatenating arrays without overlap

When concatenating arrays without overlap, the array values and the padding mask of the input arrays are not changed. But new padding bytes can be added due to data alignment. These padding bytes again get the value `in` the array and value `255` in the padding byte mask to indicate every bit is a padding bit:

```
data1:            {1, 2, 3}
data1PaddingMask: {2, 4, 8}
data1Alignment:   3
data2:            {4, 5, 6}
data2PaddingMask: {16, 32, 64}
data2Alignment:   2
Result:
{1, 2, 3, 0, 4, 5, 6} paddingMask: {2, 4, 8, 255, 16, 32, 64} alignment: 6
position in resulting array indices: data1:0 data2:4
```

3.2.4 Data Structure Padding when searching for an input array position in the resulting compressed array

When searching for an input array position in the resulting compressed array, the padding mask of the compressed array must not be taken into account. This would lead to wrong matches, as this example shows:

```
data1:            {1, 2}
data1PaddingMask: {2, 129}
data1Alignment:   1
data2:            {3, 4}
data2PaddingMask: {1, 1}
data2Alignment:   3
Result:
{0, 0, 1, 2, 3} paddingMask: {255, 255, 0, 0, 0} alignment: 3
correct position taking the resulting padding mask into account:       indices: data1:2 data2:3
incorrect position taking the resulting padding mask not into account: indices: data1:0 data2:0
```

3.3 Remove Sub-Arrays

If an input array is completely part of another input array, it is removed from the input array set (see example in section Remove Sub-Arrays). The calculation is very fast and depending on the input array set this step can already lead to a notable compression rate.

The implementation takes data alignment into account when removing sub-array as described in section Data Alignment when Removing Sub-Arrays. Please note that if there is no solution for given data-alignments, removing can fail even when the array is a sub-array.

3.4 Greedy Approach

The greedy approach merges overlapping arrays (see Greedy Approach). The implementation includes data alignment handling as described in section Data Alignment when concatenating arrays with overlap, only valid data alignment solutions are merged.

The greedy approach can be very runtime intensive for a large set of input data. Therefore the implementation pays attention to efficiency. The implementation avoids:

- unnecessary calculations and
- multiple calculations.

3.4.1 Unnecessary Calculations

Unnecessary calculations are overlap calculations which are known to fail in the first place. The implementation avoids calculations for overlap lengths which cannot occur. This optimization is shown in the next example with the following input data arrays:

```
{34, 35, 36, 37}
{20, 21, 22, 23, 24, 25}
{10, 11, 12, 13, 14, 15, 16, 17, 18, 19}
{40, 41, 42, 43, 44}
{30, 31, 32, 33, 34, 35}
```

First, the input arrays are sorted in descending order:

```
0: {10, 11, 12, 13, 14, 15, 16, 17, 18, 19} length: 10
1: {20, 21, 22, 23, 24, 25}                 length:  6
2: {30, 31, 32, 33, 34, 35}                 length:  6
3: {40, 41, 42, 43, 44}                     length:  5
4: {34, 35, 36, 37}                         length:  4
```

The algorithm starts searching for the largest possible overlap from top to bottom. In this example, the largest possible overlap is 5 which is the length of the second largest data array minus one. An overlap can't have the length of the second largest data array (=6) because this array would have been removed as sub-array in the step before. Starting with overlap 5 avoids calculations with the sizes 6-9.

On the other side the search for overlap with size 5 can stop if the array length is <=5: these arrays are too small. This avoids the calculations for the arrays 3 and 4.

In the example, an overlap of 5 is searched between arrays:
```
0 and 1
0 and 2
1 and 2
```

Since no overlap was found the overlap size is decremented to 4 resulting in a search between arrays:
```
0 and 1
0 and 2
0 and 3
1 and 2
1 and 3
2 and 3
```

And so on. Finally an overlap is found with size 2 (between arrays 3 and 4). The algorithm terminates when no overlap with size 1 is found.

When an overlap is found the concatenated array is inserted in the sorted list according to its length. This affects the search positions for overlaps but not the length of the searched overlap as seen in this example:
```
0: {9, 9, 9, 9 ,9} length: 5
1: {4, 5, 6}       length: 3
2: {5, 6, 7}       length: 3
3: {3, 4, 5}       length: 3
4: {3, 9}          length: 2
```

In this example the search for overlap is started for overlaps of length 2 between the arrays:
```
0 and 1
0 and 2
0 and 3
1 and 2
```

The arrays 1 and 2 have an overlap of 2, they are merged and added according to the merged array length:
```
0: {9, 9, 9, 9 ,9} length: 5
1: {4, 5, 6, 7}    length: 4
2: {3, 4, 5}       length: 3
3: {3, 9}          length: 2
```

Since the arrays above the insert position have already been calculated, the search starts again at the insert position. The overlap to search for is not modified, since it can't get larger by the merging arrays. So in the example the search is started again at position 1 with overlap 2.

3.4.2 Multiple Calculations

Avoiding unnecessary calculations as seen above can still lead to multiple calculations of the same overlaps when merging arrays. This can be avoided by tracking which overlaps have been already calculated. Thanks to the sorted list tracking can be reduced to just some pointers:

1. Current larger array for overlap calculation (cl)
2. Current smaller array for overlap calculation (cs)
3. first smaller array to check for left overlap (fl)
4. first smaller array to check for right overlap (fr)

Take the following example of input arrays:
```
{4, 4, 4, 4}
{0, 0, 0, 0, 0}
{2, 5}
{5, 5, 3}
{9, 9, 9, 9, 9, 9}
{1}
{5, 5, 5, 5, 5}
```

The same steps as described in section Unnecessary Calculations are performed, additionally the tracking pointers are set:
```
   cl cs fl                fr
0: ->       {9, 9, 9, 9, 9, 9}
1:    -> -> {0, 0, 0, 0, 0}    <-
2:          {5, 5, 5, 5, 5}
3:          {4, 4, 4, 4}
4:          {5, 5, 3}
5:          {2, 5}
6:          {1}
```

The pointer `cl` is set to the current main array with which the smaller arrays are checked for overlaps, this is at position 0. The first smaller array to be checked is pointed to by `cs` at position 1. The pointer `fl` and `fr` are also set to position 1, that means from position 1 up to all higher positions numbers (lower arrays in sorted list) the arrays are checked for left and right overlaps. The first overlap length to check is 4 (length of the array at position 1 minus 1).

The pointers `fl` and `fr` does not affect the search for overlaps until a merge is performed. At some point the algorithm checks for overlap 2 and an overlap is found:
```
   cl cs fl                fr
0:          {9, 9, 9, 9, 9, 9}
1:          {0, 0, 0, 0, 0}
2: ->       {5, 5, 5, 5, 5}
3:       -> {4, 4, 4, 4}       <-
4:    ->    {5, 5, 3}
5:          {2, 5}
6:          {1}
```

As described in section Unnecessary Calculations the arrays are merged and inserted in the array list again according to its length. The pointers are now adapted according to this position:
```
   cl cs fl                fr
0:          {9, 9, 9, 9, 9, 9}
1: ->       {5, 5, 5, 5, 5, 3}
2:    ->    {0, 0, 0, 0, 0}
3:          {4, 4, 4, 4}       <-
4:       -> {2, 5}
5:          {1}
```

Without taking the pointers `fl` and `fr` the algorithm would continue with overlap checks between the array at position 1 and the array at position 2. But these overlap checks have already been performed before. So an overlap of 2 is not possible between these arrays any more. It is also not possible that array at position 1 and array at position 3 have an overlap on the left side. This has also been checked before. That's why the first check for left overlaps starts with array at position 4, pointed to by `fl`.

An overlap of 2 on the right side is also not possible between arrays at position 1 and 2. This has been checked before. But since the right side of the merged array has been changed, it is possible that array at position 3 has an overlap of 2 with the merged array (e.g. assume the array at position 3 is {5, 3, 4, 4} instead of {4, 4, 4, 4}). So `fr` points to position 3.

Since no more arrays with overlap 2 are found, the algorithm now searches for overlap of size 1. The pointers are set to start positions again:
```
   cl cs fl                fr
0: ->       {9, 9, 9, 9, 9, 9}
1:    -> -> {5, 5, 5, 5, 5, 3} <-
2:          {0, 0, 0, 0, 0}
3:          {4, 4, 4, 4}
4:          {2, 5}
5:          {1}
```

At some point an overlap of size 1 is found:
```
   cl cs fl                fr
0:          {9, 9, 9, 9, 9, 9}
1: ->       {5, 5, 5, 5, 5, 3}
```

```
2:        -> {0, 0, 0, 0, 0}    <-
3:           {4, 4, 4, 4}
4:    ->     {2, 5}
5:           {1}
```

Again the arrays are merged, inserted and the pointers are set to new positions:

```
   cl cs fl                         fr
0: ->        {2, 5, 5, 5, 5, 5, 3}
1:    ->     {9, 9, 9, 9, 9, 9}
2:        -> {0, 0, 0, 0, 0}
3:           {4, 4, 4, 4}
4:           {1}                    <-
```

The first possible new left overlap is now at position 2 (assume the array is {0, 0, 0, 0, 2} instead of {0, 0, 0, 0, 0}). The first possible new right overlap is at position 4, the other arrays have been calculated before.

Depending on the input array set this tracking of already calculated overlaps avoid a lot of multiple calculations.

3.5 Result Concatenation

The result of roadC is only one data array. After removing sub-arrays and applying the greedy approach there can be still more than one array left due to:

- there is at least one array which is not a sub-array of another one,
- there are arrays without overlap, and
- the calculation was aborted by a timeout.

For concatenation the method described in section Data Alignment when concatenating arrays without overlap is implemented. The padding mask is accordingly concatenated from the padding masks of the data arrays, additionally added padding bytes by the concatenation calculation are marked as padding bytes in the resulting padding mask.

Chapter 4

Examples

The next examples show how to use the roadC library. The purpose of these examples is to provide a quick introduction into roadC and not to explain each feature in detail. Please read the reference manuel for a complete features list.

The examples are part of the release and can be complied with "`make all`".

The first example demonstrates the basic functionality of roadC. The input is a set of byte arrays with different sizes. The calculation is configured to remove sub-arrays and overlapping arrays, no timeout is given:

```
/*
The MIT License

Copyright (c) 2021 MBition GmbH

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
THE SOFTWARE.
*/
/* SPDX-License-Identifier: MIT */
/* roadC example 1 */
/* printf() */
#ifdef __cplusplus
#include <cstdio>
#else
#include <stdio.h>
#endif /* __cplusplus */
#include "roadc.h"
tRoadcByte arr1[4]={2, 3, 3, 4};
tRoadcByte arr2[3]={2, 3, 3};
tRoadcByte arr3[2]={5, 7};
tRoadcByte arr4[5]={1, 2, 3, 4, 5};
tRoadcByte arr5[3]={4, 5, 0};
int main(int arc, char **argv){
  tRoadcPtr pRoadc;
  tRoadcBytePtr compactedData;
  tRoadcUInt32 compactedDataSize;
  tRoadcUInt32 i;
  pRoadc = roadcNew();
  roadcAddElement(pRoadc, arr1, NULL, 4, 1);
  roadcAddElement(pRoadc, arr2, NULL, 3, 1);
  roadcAddElement(pRoadc, arr3, NULL, 2, 1);
  roadcAddElement(pRoadc, arr4, NULL, 5, 1);
  roadcAddElement(pRoadc, arr5, NULL, 3, 1);
  roadcCalculation(pRoadc, 1, 0);
  compactedDataSize = roadcGetCompactedDataSize(pRoadc);
  compactedData = roadcGetCompactedData(pRoadc);
  printf("Compacted data size: %lu\n", compactedDataSize);
  printf("data: ");
  for (i=0; i<compactedDataSize; i++){
    printf("%3u ", compactedData[i]);
  }
  printf("\n");
  printf("arr1 position in compacted data: %lu \n", roadcGetPositionInCompactedData(pRoadc, arr1, NULL, 4,
       1));
  printf("arr2 position in compacted data: %lu \n", roadcGetPositionInCompactedData(pRoadc, arr2, NULL, 3,
       1));
  printf("arr3 position in compacted data: %lu \n", roadcGetPositionInCompactedData(pRoadc, arr3, NULL, 2,
       1));
  printf("arr4 position in compacted data: %lu \n", roadcGetPositionInCompactedData(pRoadc, arr4, NULL, 5,
       1));
  printf("arr5 position in compacted data: %lu \n", roadcGetPositionInCompactedData(pRoadc, arr5, NULL, 3,
       1));
  roadcDelete(pRoadc);
  return 0;
}
```

The output of the first example should look like this:

```
Compacted data size: 12
data:   1   2   3   4   5   0   2   3   3   4   5   7
arr1 position in compacted data: 6
arr2 position in compacted data: 6
arr3 position in compacted data: 10
arr4 position in compacted data: 0
arr5 position in compacted data: 3
```

The second example extends the first example to data structure padding. Each input array has its own padding byte mask describing the data structure padding on bit level. This enables better compression results as in example 1:

```
/*
The MIT License

Copyright (c) 2021 MBition GmbH

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
THE SOFTWARE.
*/
/* SPDX-License-Identifier: MIT */
/* roadC example 2 */
/* printf() */
#ifdef __cplusplus
#include <cstdio>
#else
#include <stdio.h>
#endif /* __cplusplus */
#include "roadc.h"
tRoadcByte arr1[4]={2, 3, 3, 4};
tRoadcByte arr1Pad[4]={1, 2, 0, 0};
tRoadcByte arr2[3]={2, 3, 3};
tRoadcByte arr2Pad[3]={0, 255, 0};
tRoadcByte arr3[2]={5, 7};
tRoadcByte arr3Pad[2]={0, 252};
tRoadcByte arr4[5]={1, 2, 3, 4, 5};
tRoadcByte arr4Pad[5]={255, 255, 0, 0, 0};
tRoadcByte arr5[3]={4, 5, 0};
tRoadcByte arr5Pad[3]={0, 0, 255};
int main(int arc, char **argv){
  tRoadcPtr pRoadc;
  tRoadcBytePtr compactedData;
  tRoadcBytePtr compactedDataPaddingByteMask;
  tRoadcUInt32 compactedDataSize;
  tRoadcUInt32 i;
  pRoadc = roadcNew();
  roadcAddElement(pRoadc, arr1, arr1Pad, 4, 1);
  roadcAddElement(pRoadc, arr2, arr2Pad, 3, 1);
  roadcAddElement(pRoadc, arr3, arr3Pad, 2, 1);
  roadcAddElement(pRoadc, arr4, arr4Pad, 5, 1);
  roadcAddElement(pRoadc, arr5, arr5Pad, 3, 1);
  roadcCalculation(pRoadc, 1, 0);
  compactedDataSize = roadcGetCompactedDataSize(pRoadc);
  compactedData = roadcGetCompactedData(pRoadc);
  compactedDataPaddingByteMask = roadcGetCompactedDataPaddingByteMask(pRoadc);
  printf("Compacted data size: %lu\n", compactedDataSize);
  printf("data: ");
  for (i=0; i<compactedDataSize; i++){
    printf("%3u ", compactedData[i]);
  }
  printf("\nmask: ");
  for (i=0; i<compactedDataSize; i++){
    printf("%3u ", compactedDataPaddingByteMask[i]);
  }
  printf("\n");
  printf("arr1 position in compacted data: %lu \n", roadcGetPositionInCompactedData(pRoadc, arr1, arr1Pad,
       4, 1));
  printf("arr2 position in compacted data: %lu \n", roadcGetPositionInCompactedData(pRoadc, arr2, arr2Pad,
       3, 1));
  printf("arr3 position in compacted data: %lu \n", roadcGetPositionInCompactedData(pRoadc, arr3, arr3Pad,
       2, 1));
  printf("arr4 position in compacted data: %lu \n", roadcGetPositionInCompactedData(pRoadc, arr4, arr4Pad,
       5, 1));
  printf("arr5 position in compacted data: %lu \n", roadcGetPositionInCompactedData(pRoadc, arr5, arr5Pad,
       3, 1));
  roadcDelete(pRoadc);
  return 0;
}
```

The output of the second example should look like this:

```
Compacted data size: 9
data:   2   5   3   4   5   2   3   3   4
mask:   0   0   0   0   0   1   2   0   0
arr1 position in compacted data: 5
arr2 position in compacted data: 0
arr3 position in compacted data: 1
arr4 position in compacted data: 0
arr5 position in compacted data: 3
```

The third example introduces data alignment handling. The memory alignments are given as parameters in the functions roadcAddElement and roadcGetPositionInCompactedData. These restrictions result in a larger compressed data size than in example 2:

```
/*
The MIT License

Copyright (c) 2021 MBition GmbH

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
THE SOFTWARE.
*/
/* SPDX-License-Identifier: MIT */
/* roadC example 3 */
/* printf() */
#ifdef __cplusplus
#include <cstdio>
#else
#include <stdio.h>
#endif /* __cplusplus */
#include "roadc.h"
tRoadcByte arr1[4]={2, 3, 3, 4};
tRoadcByte arr1Pad[4]={1, 2, 0, 0};
tRoadcByte arr2[3]={2, 3, 3};
tRoadcByte arr2Pad[3]={0, 255, 0};
tRoadcByte arr3[2]={5, 7};
tRoadcByte arr3Pad[2]={0, 252};
tRoadcByte arr4[5]={1, 2, 3, 4, 5};
tRoadcByte arr4Pad[5]={255, 255, 0, 0, 0};
tRoadcByte arr5[3]={4, 5, 0};
tRoadcByte arr5Pad[3]={0, 0, 255};
int main(int arc, char **argv){
  tRoadcPtr pRoadc;
  tRoadcBytePtr compactedData;
  tRoadcBytePtr compactedDataPaddingByteMask;
  tRoadcUInt32 compactedDataSize;
  tRoadcUInt32 compactedDataAlignment;
  tRoadcUInt32 i;
  pRoadc = roadcNew();
  roadcAddElement(pRoadc, arr1, arr1Pad, 4, 1);
  roadcAddElement(pRoadc, arr2, arr2Pad, 3, 4);
  roadcAddElement(pRoadc, arr3, arr3Pad, 2, 8);
  roadcAddElement(pRoadc, arr4, arr4Pad, 5, 4);
  roadcAddElement(pRoadc, arr5, arr5Pad, 3, 2);
  roadcCalculation(pRoadc, 1, 0);
  compactedDataSize = roadcGetCompactedDataSize(pRoadc);
  compactedDataAlignment = roadcGetCompactedDataAlignment(pRoadc);
  compactedData = roadcGetCompactedData(pRoadc);
  compactedDataPaddingByteMask = roadcGetCompactedDataPaddingByteMask(pRoadc);
  printf("Compacted data size: %lu alignment: %lu\n", compactedDataSize, compactedDataAlignment);
  printf("data: ");
  for (i=0; i<compactedDataSize; i++){
    printf("%3u ", compactedData[i]);
  }
  printf("\nmask: ");
  for (i=0; i<compactedDataSize; i++){
    printf("%3u ", compactedDataPaddingByteMask[i]);
  }
  printf("\n");
  printf("arr1 position in compacted data: %lu \n", roadcGetPositionInCompactedData(pRoadc, arr1, arr1Pad,
       4, 1));
  printf("arr2 position in compacted data: %lu \n", roadcGetPositionInCompactedData(pRoadc, arr2, arr2Pad,
       3, 4));
  printf("arr3 position in compacted data: %lu \n", roadcGetPositionInCompactedData(pRoadc, arr3, arr3Pad,
       2, 8));
  printf("arr4 position in compacted data: %lu \n", roadcGetPositionInCompactedData(pRoadc, arr4, arr4Pad,
       5, 4));
  printf("arr5 position in compacted data: %lu \n", roadcGetPositionInCompactedData(pRoadc, arr5, arr5Pad,
       3, 2));
  roadcDelete(pRoadc);
  return 0;
}
```

The output of the third example should look like this:

```
Compacted data size: 10 alignment: 8
data:   0   0   4   5   2   1   3   4   5   7
mask: 255 255   0   0   0   2   0   0   0 252
arr1 position in compacted data: 4
arr2 position in compacted data: 4
arr3 position in compacted data: 8
arr4 position in compacted data: 4
arr5 position in compacted data: 2
```

Chapter 5

License