Compression Algorithms DAN0

Presented by Daniel Bienvenu

Last revision : March 21, 2010

Resume : DAN0 algorithm is a Run Length Encoding with a Fixed Huffman Encoding pointing to a not fixed window of 20 bytes data, partially acting like a buffer, without using extra temporary memory. The Run Length Encoding (RLE) part reduces the size of consecutive bytes all the same value, which happens quite often within graphics data, by saying first how many times this value is copied in the uncompressed sequence. The Fixed Huffman Encoding part saves a few bits for each byte already available in a window of 20 bytes representing data already encoded. And a variation of the same algorithm is proposed with speed and size advantages. It was tested with real graphics for TMS9928a video chip machines like the ColecoVision.

Table of content

Introduction........3
  Run Length Encoding........3
  Fixed Huffman Encoding........3
Encoding control bytes and bits........5
Optimization strategies........6
  Strategy #1 - Storing data........6
  Strategy #2 - Stealing data........6
Testing with real cases........7
Can it be better?........8
  First suggestion – changing the storage part........8
  Second suggestion – changing the Huffman part........8
  Third suggestion – changing the parameters........8
  Last suggestion – try all the above........8
DAN0 – a smaller variation........9
  New Run Length Encoding (RLE) table........9
  New fixed Huffman table........9
  Compare routines size........9
Testing the alternative version with real cases........10
Conclusion........11
APPENDIX - 1........12
APPENDIX - 2........15
APPENDIX - 3........18

Introduction

DAN0 algorithm is a compression algorithm created by Daniel Bienvenu in year 2010 based on saving space while encoding empty spaces, repetitions, and bytes already encoded. To do that, the algorithm is a combination of 2 ways of encoding data : simple Run Length Encoding (RLE) saves empty spaces, a Fixed Huffman Encoding avoid encoding some repetitions.

DAN0 algorithm needs 2 tables : control table and data table. The control table is used to encode compression codes. The data table is used to encode values from the original sequence but based on the compression encoded in the control table.

Run Length Encoding

The Run Length Encoding part of DAN0 is the main part of the algorithm, that's why it needs a special code telling that the compression part stops there. This special byte is value 0. Notice that this special byte 0 will be always at the end of the control bytes table. All the other values are explained in the following table.

Byte value	Description
0	End
1 to 126	2 to 127 copies of a single byte
127	256 copies of a single byte
128	256 bytes to copy
129 to 255	1 to 127 bytes to copy

Let's talk about the Fixed Huffman encoding part.

Fixed Huffman Encoding

The Fixed Huffman Encoding part of DAN0 tries to save more bytes than using a straight RLE algorithm. To do that, and to avoid a too complicated unpacking routine, a fixed Huffman table is used as follow to calculate an index. This index points in a window of data to a byte to output. This window of data change its position inside the data table if the byte to output is not present in the window already. Because this part may need more bits than a straight RLE encoding, a simple RAW storage of data can be used instead. The data table starts at a marker byte 0 for a storage encoding strategy; it's Huffman encoding strategy otherwise. And if the Huffman encoding starts with byte 0 to be stored in the data table, then arrange tables in a way to get a byte 0 before the data table.

Huffman value in bits	**Description**
0	Move window pointer one byte and read its new last byte
1000	Read last byte (same as the 20th one) in window
1010	Read 2nd last byte (same as the 19th one) in window
1100	Read 3rd last byte (same as the 18th one) in window
1110	Read 4th last byte (same as the 17th one) in window
100100	Read 5th last byte (same as the 16th one) in window
100101	Read 6th last byte (same as the 15th one) in window
100110	Read 7th last byte (same as the 14th one) in window
100111	Read 8th last byte (same as the 13th one) in window
101100	Read 9th last byte (same as the 12th one) in window
101101	Read 10th last byte (same as the 11th one) in window
101110	Read 11th last byte (same as the 10th one) in window
101111	Read 12th last byte (same as the 9th one) in window
110100	Read 13th last byte (same as the 8th one) in window
110101	Read 14th last byte (same as the 7th one) in window
110110	Read 15th last byte (same as the 6th one) in window
110111	Read 16th last byte (same as the 5th one) in window
111100	Read 17th last byte (same as the 4th one) in window
111101	Read 18th last byte (same as the 3rd one) in window
111110	Read 19th last byte (same as the 2nd one) in window
111111	Read 20th last byte (same as the 1st one) in window

Let's show an example of possible Huffman encoding

Let's encode the word : **ARCADE**

A – never seen before, a new byte to encode : Huffman “0”, Data “A”.
R – never seen before, a new byte to encode : Huffman “00”, Data “AR”.
C – never seen before, a new byte to encode : Huffman “000”, Data “ARC”.
A – seen before, as the 3rd last byte in the window : Huffman “0001100”, Data “ARC”.
D – never seen before, a new byte to encode : Huffman “00011000”, Data “ARCD”.
E – never seen before, a new byte to encode : Huffman “000110000”, Data “ARCDE”.

Here, the algorithm avoid storing twice the character A in the data table, saving then 1 byte. However, to do that, the Huffman encoding part needed 9 bits, stored into more than 1 byte. Considering that the Huffman codes here needed more bytes than it helps to save for the data part, the algorithm will simply avoid using the Huffman encoding strategy and store the data unchanged instead. This situation occurs often when data can't be compressed well with this compression algorithm.

Encoding control bytes and bits

The RLE codes are encoded to the control table as soon as we calculated them. The marker for the end of data is byte value 0. The Huffman codes are encoded into bytes to the control table, and each byte is added to the control table as soon as we need them, but we keep updating the same byte before adding a new byte to the control table to continue the Huffman encoding. And when there is no more Huffman codes to encode, the last byte with Huffman codes in it is completed with 0s for the lower bits.

Example:

> Let's encode : WOOOOOOW!
>
> PASS #1 – RLE
>
> [1, “W”], [6, “O”], [1, “W”], [1, “!”]
>
> RLE = [129,5,129,129,0]
> DATA = [0] + “WOW!”
>
> PASS #2 – HUFFMAN
>
> [“0”, “W”], [“0”, “O”], [“1010”], [“0”, “!”]
>
> HUFFMAN = “0010100” = decimal value 40 by adding an extra bit 0 to complete a byte.
> DATA = “WO!”
>
> PASS #3 – MERGING RLE AND HUFFMAN
>
> CONTROL = [129, 40, 5, 129, 129, 0]
> DATA = “WO!”

Optimization strategies

Strategy #1 - Storing data

To know if the data table is simple storage (not part of Huffman encoding strategy), we need a special marker. This marker is a byte value 0 at the beginning of the data table. Instead of adding a byte, reuse the last byte of the control table; always ends the special marker byte 0 for the end of compression.

Example :

> *CONTROL = [134,0]; and DATA = [0] + "ARCADE".*
> **Total** = <u>9 bytes</u>

Become

> *CONTROL = [134,0] + "ARCADE", where DATA starts at byte 0.*
> **Total** = <u>8 bytes</u>, saving 1 byte.

Strategy #2 - Stealing data

To save extra bytes while using Huffman encoding strategy, you can reuse bytes from the end of another data table to save a few bits for at least the first bytes to encode. This strategy can also help to save bytes by using Huffman Encoding instead of simple storage in some cases.

Example :

We want to encode ARCADE and DANCE. Both cases need normally the storage strategy. However, these words have letters in common, so we can expect using Huffman encoding to save bytes. We can store DANCE and get letters D,A,C and E from it to encode ARCADE.

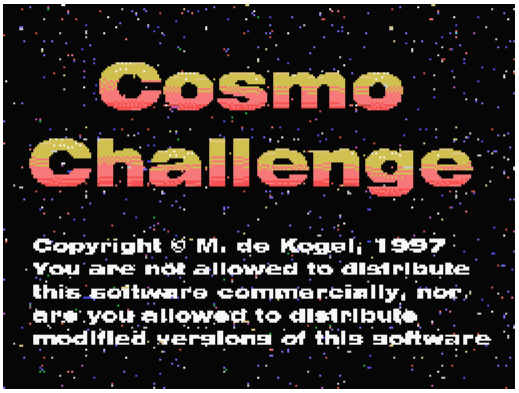
> *CONTROL1 = [133,0]+"DANCE", DATA1 points to byte value 0.*
> *CONTROL2 = [134,0]+"ARCADE", DATA2 points to byte value 0.*
> **Total** = <u>15 bytes</u>

Become

> *CONTROL1 = [133,0]+ "DANCER", DATA1 points to byte 0, DATA2 points to "R".*
> *CONTROL2 = [134, 230, 73, 45, 0, 0].*
> **Total** = <u>14 bytes</u>, saving 1 byte.

Testing with real cases

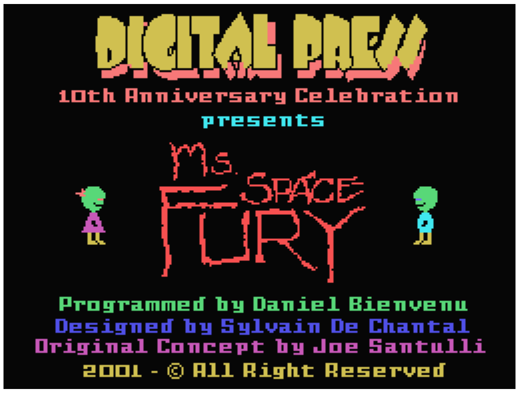
The following pictures are from real projects, and need exactly 12288 bytes (12 kilo-bytes or 12 KB). The first number beside the word “RLE” is the compressed data size in bytes used in these projects, and the second number is by adding the size of the decompression routine. Please note that the RLE algorithm used was done by Marcel de Kogel in 1996 and distributed freely. The first number beside the word “DAN0” is the compressed data size resulting of the new algorithm proposed in this paper, the second number is by adding the size of the decompression routine. Even if the decompression routine is bigger, DAN0 gives a better compression overall.


Cosmo Challenge
Copyright © M. de Kogel, 1997
You are not allowed to distribute this software commercially, nor are you allowed to distribute modified versions of this software

ColecoVision Cosmo Challenge (game)
by Marcel de Kogel, 1997

RLE : 7720, 7774 (63.3%)

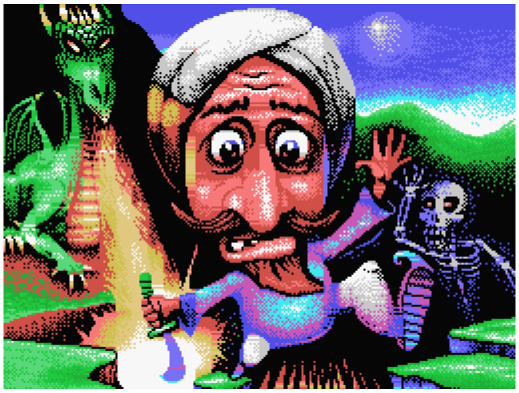
DAN0 : 6617, 6746 (54.9%)


DIGITAL PRESS
10th Anniversary Celebration
presents
Ms. SPACE FURY
Programmed by Daniel Bienvenu
Designed by Sylvain De Chantal
Original Concept by Joe Santulli
2001 - © All Right Reserved

ColecoVision Ms Space Fury (game)
by Daniel Bienvenu, 2001

RLE : 3382, 3436 (28.0%)

DAN0 : 2683, 2811 (22.9%)



Turban (from MSX demo : Alankomaat)
by Mermaid, 2000

RLE : 8423, 8477 (69.0%)

DAN0 : 7796, 7924 (64.5%)

Can it be better?

Yes, it can be better depending of your needs. Keep in mind that this paper is a proposition of a compression algorithm. You can modify it for your own needs, making it a more personal version. This is a few suggestions to reduce the routine size and making it even faster.

First suggestion – changing the storage part

You can save bytes and speed up the decompression routine by cutting the storage option to keep only the Huffman part. Of course, make sure that you'll not need this storage option before thinking of using this solution. In general, bigger is the data to compress, better is the result with Huffman encoding. An alternative could be to replace the storage part by another strategy which will not reduce the size of the routine but can be justified for your needs depending of the data.

Second suggestion – changing the Huffman part

You can change the fixed Huffman table part by another one or by something different like a gamma table (a strategy used in BitBuster and PuCrunch). After a few tests with the 12 kilo-bytes pictures, it looks like I should use a different fixed Huffman table to get sometimes better results. By using a more modest Huffman table, the decoding part inside the routine becomes smaller and faster.

Third suggestion – changing the parameters

This is for the C programmers. It's possible to reduce the number of pop and push instructions, some of them are inside the routine, and the others are generated for each call to this routine to setup the parameters. All this can be reduced by cutting the number of parameters, but you still need the two tables (control and data). The solution is to simply put the pointer to the data table inside the control table and use only the control table as a parameter. This will save a few bytes in your code.

Last suggestion – try all the above

All these strategies seems too good to be true. So, I've decided to try them all for an extreme version of my algorithm. First, I did cut the number of parameters, which didn't reduce the number of bytes needed inside the routine, but certainly cut the number of bytes needed for the calls to this routine. I did cut the storage option part to keep only the Huffman encoding part, which did save a bunch of bytes, making the routine smaller and faster. And finally, I've decided to use a different fixed Huffman table which did reduce by 2 bytes the decoder part, making it again smaller and faster.

DAN0 – a smaller variation

The following version don't use the storage option, so no need for the marker byte value 0, to focus on Huffman encoding to save bytes. It uses a different RLE table to optimize the decompression routine and a different fixed Huffman table that seems to give a better compression overall.

New Run Length Encoding (RLE) table

Byte value	**Description**
0	256 bytes to copy
1 to 127	1 to 127 bytes to copy
128	256 copies of a single byte
129	END
130 to 255	2 to 127 copies of a single byte

New fixed Huffman table

This table focus on the 2, instead of 4, recent added bytes as the most commonly used ones.

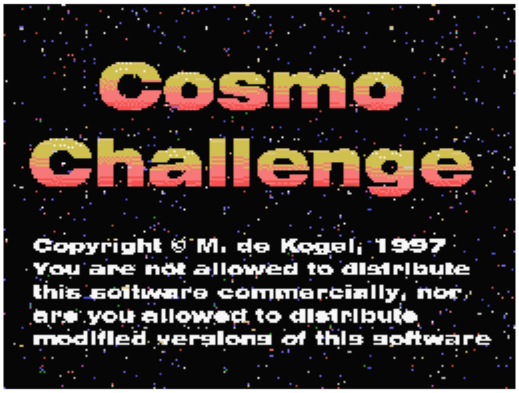
Huffman value in bits	**Description**
0	Move window pointer one byte and read its new last byte
100	Read last byte (same as the 10th one) in window
110	Read 2nd last byte (same as the 9th one) in window
10100	Read 3rd last byte (same as the 8th one) in window
10101	Read 4th last byte (same as the 7th one) in window
10110	Read 5th last byte (same as the 6th one) in window
10111	Read 6th last byte (same as the 5th one) in window
11100	Read 7th last byte (same as the 4th one) in window
11101	Read 8th last byte (same as the 3rd one) in window
11110	Read 9th last byte (same as the 2nd one) in window
11111	Read 10th last byte (same as the 1st one) in window

Compare routines size

The new routine is way smaller than the original one, making it faster too. Also, the new source code includes the assembler entry point if needed. This is reduction in size was possible by cutting the number of condition branches, pop & push usage, and a few other instructions. Now, the question is : Does it compress well?

Testing the alternative version with real cases

The following pictures are from real projects, and need exactly 12288 bytes (12 kilo-bytes or 12 KB). This time, we compare the compressed data size with and without the decompression routine size for 3 algorithms : “RLE” by Marcel de Kogel, “DAN0” by Daniel Bienvenu, and “DAN0 Alternative” also by Daniel Bienvenu. Even if the decompression routine is bigger, both DAN0 algorithms are giving a better compression overall. And yes, cutting the storage option part and changing the fixed Huffman table have a good impact on the compression ratio.



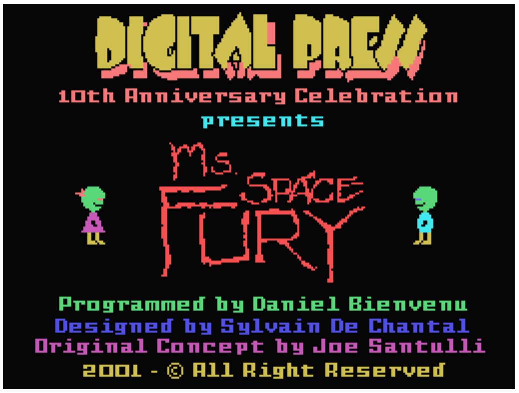
Copyright © M. de Kogel, 1997
You are not allowed to distribute
this software commercially, nor
are you allowed to distribute
modified versions of this software

ColecoVision Cosmo Challenge (game)
by Marcel de Kogel, 1997

RLE : 7720, 7774 (63.3%)

DAN0 : 6617, 6746 (54.9%)

DAN0[alt] : 6271, 6372 (51.9%)


DIGITAL PRESS
10th Anniversary Celebration
presents
Ms. SPACE FURY
Programmed by Daniel Bienvenu
Designed by Sylvain De Chantal
Original Concept by Joe Santulli
2001 - © All Right Reserved

ColecoVision Ms Space Fury (game)
by Daniel Bienvenu, 2001

RLE : 3382, 3436 (28.0%)

DAN0 : 2683, 2811 (22.9%)

DAN0[alt] : 2675, 2776 (22.6%)

Turban (from MSX demo : Alankomaat)
by Mermaid, 2000

RLE : 8423, 8477 (69.0%)

DAN0 : 7796, 7924 (64.5%)

DAN0[alt] : 7757, 7858 (63.9%)

Conclusion

DAN0 is a compression algorithm that offers an alternative to the straight Run Length Encoding (RLE) without using much resources compared to other complex algorithms already existing like PuCrunch, BitBuster, Pletter, etc. Because DAN0 is made in a way to focus on vintage consoles and computers, even a few saved bytes can be seen as a good fortune to be able to give more for the same size of data. And the best part, it doesn't use extra memory in your projects to do its job. Depending on the project, saving a bunch of bytes can be enough to improve graphics and animations, levels and gameplay, and so on. Of course, DAN0 is slower than RLE, but it's fast enough to decompress data in no time on vintage machines like the ColecoVision.

I suggest to implement the proposed DAN0 algorithms in your projects if you're looking for a way to use compressed data without the need for extra RAM and still get a better compression than RLE. Please do credit my works if you do use DAN0 or talk about it. Thanks a lot for reading and have fun!

- Daniel Bienvenu, March 2010

APPENDIX - 1

Listing – DAN0 decoder in Z80 assembly codes

```
    ;;  HL = Pointer to CONTROL table, DE = Pointer to DATA table
    ;;  C = I/O port to output uncompressed data
;; DAN0 entry point routine
dan0:
    ld a,#0x80
    ex af,af'  ;; Set register A' with a marker to read properly Huffman encoded bits.
    ld  a,(de)
    or a ;;  If first byte in data table is marker 0, then it's storage strategy
    push af ;; Z flag stored in stack
    jr nz, dan0_main
    inc de ;;  and increase data table pointer to the real first byte to read

;; Run Length Encoding (main loop)
dan0_main:
    ld a,(hl) ;; read next byte in control table
    inc hl
    bit 7,a
    jr nz, dan0_raw ;; swicth to raw or rle encoding
    or a
    jr z, dan0_end ;; jump if it's marker byte 0 in control table
    inc a
    and #0x7f
    ld b,a
    pop af
    push af
    call dan0_readnextbyte
dan0_rle_loop:
    out (c),a
    djnz dan0_rle_loop
    jr dan0_main
dan0_raw:
    and #0x7f
    ld b,a
dan0_raw_loop:
    pop af
    push af
    call dan0_readnextbyte
    out (c),a
    djnz dan0_raw_loop
    jr dan0_main
```

```
;; EXIT
dan0_end:
   pop af ;; restore AF register pair before leaving
   ret

;; Read next byte in Data table according to the encoding strategy
dan0_readnextbyte:
   jr  nz, dan0_compression
   ld a,(de)
   inc de
   ret
;; Huffman Encoding Strategy
dan0_compression:
   ex af,af'
   push bc ;; save register pair BC
   ld bc,#0x0000
   call dan0_getbit
   ex de,hl
   jr nc, dan0_getbyte
   ex de,hl
   call dan0_getbit
   call dan0_rlcgetbit
   call dan0_rlcgetbit
   jr nc, dan0_set_buffer_ptr
   call dan0_getbit
   call dan0_rlcgetbit
   rl c
   inc c
   inc c
   inc c
   inc c
dan0_set_buffer_ptr:
   inc c
   ex de,hl
   sbc hl,bc
   dec c
dan0_getbyte:
   ex af,af'
   ld a,(hl)
   inc hl
   add hl,bc
   ex de,hl
   pop bc
   ret
```

```
;; Routines to get the next single bit of information
dan0_rlcgetbit:
   rl c
dan0_getbit: ;; get next bit in control table
   add a,a
   ret nz
   ld a,(hl)
   inc hl
   rla
   ret
```

APPENDIX - 2

Listing – DAN0 decoder in Z80 assembly codes for ColecoVision projects in C language.

*Usage – dan0(unsigned vram_offset, void *tbldata, void *tblcodes);*

```
;-------------------------------------------------------------------------------
; DAN0 VRAM Depacker v1.0 by Daniel Bienvenu, March 2010.
;-------------------------------------------------------------------------------
;  void dan0(unsigned vram_offset, void *tbldata, void *tblcodes);
;-------------------------------------------------------------------------------
; Compiled size = 129 bytes.
; I/O ports BFh (control) and BEh (data) are for Video RAM access.
;-------------------------------------------------------------------------------
    .module dan0
    .globl  _dan0
    .area _CODE

_dan0:
    pop hl
    pop de
    ld  c,#0xbf
    out (c),e
    set 6,d
    out (c),d
    pop de
    pop bc
    push bc
    push de
    push de
    push hl
    ld h,b
    ld l,c
    ld c,#0xbe
    ld a,#0x80
    ex af,af'
    ld  a,(de)
    or a
    push af ;; push flag Z
    jr nz, dan0_main
    inc de

    ;;  DE = tbldata ptr, HL = tblcodes ptr
    ;;  A' = 0x80, C = 0xBE, Z FLAG PUSHED
```

```
dan0_main:
   ld a,(hl)
   inc hl
   bit 7,a
   jr nz, dan0_raw
   or a
   jr z, dan0_end
   inc a
   and #0x7f
   ld b,a
   pop af
   push af
   call dan0_readnextbyte

dan0_rle_loop:
   out (c),a
   djnz dan0_rle_loop
   jr dan0_main

dan0_raw:
   and #0x7f
   ld b,a

dan0_raw_loop:
   pop af
   push af
   call dan0_readnextbyte
   out (c),a
   djnz dan0_raw_loop
   jr dan0_main

dan0_end:
   pop af
   ret

;; Read next byte in Data table according to the encoding strategy
dan0_readnextbyte:
   jr  nz, dan0_compression

;; Read not compressed data
   ld a,(de)
   inc de
   ret
```

```
;; Huffman decoding
dan0_compression:
   push bc
   ld bc,#0x0000
   ex af,af'
   call dan0_getbit
   ex de,hl
   jr nc, dan0_getbyte
   ex de,hl
   call dan0_getbit
   call dan0_rlcgetbit
   call dan0_rlcgetbit
   jr nc, dan0_set_buffer_ptr
   call dan0_getbit
   call dan0_rlcgetbit
   rl c
   inc c
   inc c
   inc c
   inc c
dan0_set_buffer_ptr:
   inc c
   ex de,hl
   sbc hl,bc
   dec c

dan0_getbyte:
   ex af,af'
   ld a,(hl)
   add hl,bc
   ex de,hl
   inc de
   pop bc
   ret

dan0_rlcgetbit:
   rl c
dan0_getbit:
   add a,a
   ret nz
   ld a,(hl)
   inc hl
   rla
   ret
```

APPENDIX - 3

Listing – DAN0 alternative decoder in Z80 assembly codes for ColecoVision projects in C language.

*Usage – dan0alt(unsigned vram_offset, void *table);*

```
;-------------------------------------------------------------------------------
; DAN0 Alternative VRAM Depacker v1.0 by Daniel Bienvenu, March 2010.
; Compiled size = 101 bytes.
;-------------------------------------------------------------------------------
;  USAGE : void dan0alt(unsigned vram_offset, void *table);
;-------------------------------------------------------------------------------
; RLE codes (different than original DAN0)
; 00h = 256 bytes to output
; 01h-7Fh = 1 to 127 bytes to output
; 80h = 256 times 1 byte to output
; 81h = END
; 82h-FFh = 2 to 127 times 1 byte to output
;-------------------------------------------------------------------------------

   .module dan0
   .globl  _dan0alt, dan0alt
   .area _CODE

;; C routine entry point,  In stack = Return Pointer, VRAM Offset, Table Pointer
_dan0alt:
   pop bc
   pop de
   pop hl
   push hl
   push de
   push bc

;; ASM routine entry point
;; IN : HL = VRAM OFFSET , DE = CTRL_TABLE POINTER - 2 ( data_table pointer stored in )
;; OUT : HL = POINTER TO AFTER THE END OF DATA_TABLE
;;           DE = POINTER TO AFTER THE END OF CTRL_TABLE
;;           C = I/O PORT ( = #0xbe)  ;  AF, AF' and B = GARBAGE

dan0alt:
   ; OFFSET VRAM
   ld  c,#0xbf
   out (c),l
   set 6,h
   out (c),h
   ; SET DATA_TABLE POINTER IN DE ( FROM TABLE SET IN HL)
   ex de,hl
   ld e,(hl)
   inc hl
   ld d,(hl)
   inc hl
   ex de,hl
   dec c  ; ld c,#0xbe

   ld a,#0x80
   ex af,af'

   ;;  DE = CTRL_TABLE POINTER
   ;;  HL = DATA_TABLE POINTER
   ;;  A' = 0x80, C = I/O PORT (0xBE)
dan0_main:
   ld a,(de)
   inc de
   bit 7,a
   jr z, dan0_raw
   and #0x7f
   ld b,a
   sub #1
   ret z ; END OF DECOMPRESSION

  ;;  RLE CASE
   call dan0_readnextbyte

dan0_rle_loop:
   out (c),a
   djnz dan0_rle_loop
   jr dan0_main

  ;;  RAW CASE
dan0_raw:
   ld b,a

dan0_raw_loop:
   call dan0_readnextbyte
   out (c),a
   djnz dan0_raw_loop
   jr dan0_main
```

```
;; Read next byte is always Huffman decoder routine
dan0_readnextbyte:
    push bc
    ld bc,#0x0000
    ex af,af'
    call dan0_getbit
    jr nc, dan0_getbyte
    call dan0_getbit
    call dan0_rlcgetbit
    jr nc, dan0_set_buffer_ptr
    call dan0_getbit
    call dan0_rlcgetbit
    rl c
    inc c
    inc c
dan0_set_buffer_ptr:
    inc c
    sbc hl,bc
    dec c

dan0_getbyte:
    ex af,af'
    ld a,(hl)
    add hl,bc
    inc hl
    pop bc
    ret

dan0_rlcgetbit:
    rl c

dan0_getbit:
    add a,a
    ret nz
    ld a,(de)
    inc de
    rla
    ret
```