**10LINES ADV**

**10LINES ADV** is an adventure game for [COMMODORE 64](https://it.wikipedia.org/wiki/Commodore_64) home computer. In this game the player assumes the role of a protagonist in an interactive story, driven by exploration and puzzle-solving. This game is designed for a single player, it is text-based and uses a text parser to translate the player's input into commands.

[Click here](https://github.com/spotlessmind1975/adv10en/blob/master/docs/instructions.md) to read the instructions on how to play, and [here](https://github.com/spotlessmind1975/adv10en/blob/master/adv10-all.d64) to download the disk image (D64 format). The image contains both English and [Italian edition](https://github.com/spotlessmind1975/adv10).

[](https://github.com/spotlessmind1975/adv10en/blob/master/snapshot.png)

This game was created to participate in the ["BASIC10Liner" competition 2020](https://gkanold.wixsite.com/homeputerium/2020). For this reason, the game is modest in size (even if it has a couple of puzzles), the given source code respects strong constraints on the length of each line (category: **EXTREME-256**) and it does not rely on third party libraries. For those interested, the Italian edition is also available.

For those wishing to deepen the details of its implementation (**ATTENTION: SPOILERS!**), please refer to the following documents:

* [the source code](https://github.com/spotlessmind1975/adv10en/blob/master/docs/source.md)
* [the parser](https://github.com/spotlessmind1975/adv10en/blob/master/docs/parser.md)
* [the internal state of the game](https://github.com/spotlessmind1975/adv10en/blob/master/docs/game-state.md)
* [the compression method](https://github.com/spotlessmind1975/adv10en/blob/master/docs/compression.md)

For those who are stuck, find suggestions [on this page](https://github.com/spotlessmind1975/adv10en/blob/master/docs/suggestions.md). Here you will find the [complete solution](https://github.com/spotlessmind1975/adv10en/blob/master/docs/solution.md).

**INSTRUCTIONS**

In order to play "10lines Adventure" you need a COMMODORE 64 or an emulator of this computer, because the game does not work on modern computers. The game has been tried and it works correctly with *"VICE - the Versatile Commodore Emulator" ver. 3.2*. You can download it for free by [clicking here](http://vice-emu.sourceforge.net/).

Once the emulator software has been installed, follow the instructions below:

* start **x64.exe** program;
* click on **File** > **Attach disk image** > **Drive 8**;
* select adv10-all.d64 file ([click here to download](https://github.com/spotlessmind1975/adv10en/blob/master/adv10-all.d64));
* click on **Attach**;
* type LOAD "ADV10EN",8,1;
* when finished loading, type RUN

# SOURCE CODE (EXPLAINED)

Below you will find the source code of the video game. The source code has been written extensively (without abbreviations), in order to make it easier to understand. Each line has been commented to illustrate how the code works.

## INITIALIZATION (LINES 1-3)

The order of the lines in this section is from the shortest to the longest row. In this way, we will optimize the space available in the first three lines of code.

1 x=0

Let's set the room (**x**) to zero. Later we will modify it automatically, or based on the actions performed by the player. See also [the current room (X)](https://github.com/spotlessmind1975/adv10en/blob/master/docs/game-state.md#the-current-room-x)

t=48

This variable will store the character '0' as a PETSCII code. This is an optimization when poking the value on the screen, and is not directly related to the game.

print "{clear}";

Let's clear the screen (we do it only once!).

k=1061 : n=55333

Pointer to the beginning of the character video memory (k) and color video memory (n). This is where we are going to print the score, or the 37th column on the first row. On the Commodore 64, the text video memory starts from the position 1024 and the color video memory starts from 55296, so: 1024 + 37 = 1061; 55296 + 37 = 55333. This is an optimization when poking values on the screen, and is not directly related to the game.

w=54272

Pointer to the beginning of the audio registers. This is an optimization when poking the value on the audio controller, and is not directly related to the game.

dim d$(17)

From this point on, we declarate the texts that pertain to the various components of the game (for a total of 17 texts): descriptions of the rooms, objects, error messages, titles, and so on. All texts are stored in a one-dimensional array, the index of which is calculated. The index schema guarantees an easy accessibility by means of a basic calculation. For example, the room descriptions coincide almost with the room number. The actual texts are compressed using the ["nibble compression" technique](https://github.com/spotlessmind1975/adv10en/blob/master/docs/compression.md).

poke 19,32

Change the current I/O device number to 32. Following [this guide](https://www.c64-wiki.com/wiki/Zeropage), only the first 5 bits are used. So, by setting 32, we are going to set again the keyboard but using it as a "file". This wil disable the question mark (?) print out on keyboard input - we will replace it with an explicit and more classic "more-than" symbol (>).

poke 53281,0 : poke 53280,0

Now we set the background and border color of the screen ([here for a complete map](https://www.c64-wiki.com/wiki/Page_208-211)). The best is to use the black color (0) for both. This is "classic" color of text adventures.

d$(11)="{236}{075}"

Message: ok.

d$(10)="{094}{085}{238}{077}{063}"

Message: uhm?

d$(6)="{206}{089}{254}{085}{228}{088}{029}{148}{254}{033}"

Message: you exited!

:d$(8)="{206}{089}{254}{085}{067}{244}{242}{202}{078}{080}"

Message: you see a rope

d$(14)="{203}{081}{189}{254}{071}{206}{077}{074}{247}"

Message: nothing more.

d$(15)="{081}{244}{201}{172}{223}{243}{104}{060}{148}{247}"

Message: the door is closed.

d$(4)="{236}{086}{164}{031}{069}{239}{080}{148}{052}{033}{118}"

Message: over the pedestal.

v$="tasehl.cdrnoi\*\*\*"

Here we find the "dictionary" used for [compression](https://github.com/spotlessmind1975/adv10en/blob/master/docs/compression.md). The sequence of characters has been carefully chosen, so that escape sequences are kept to a minimum. Note that only the first thirteen (13) characters are used by the encoding; the rest are filled with arbitrary characters ("\*"). The 16-character extension is required to avoid overflow situations in the decompression routine, when escape or space nibbles are found.

d$(5)="{189}{047}{031}{172}{225}{085}{074}{143}{037}{238}{077}{066}{164}{247}"

Message: in a torture chamber.

2 d$(12)="{238}{042}{042}{239}{049}{110}{048}{189}{052}{047}{233}{086}{239}{042}{254}{042}"

Message: \*\* 10lines adv \*\*

d$(7)="{206}{089}{254}{085}{040}{235}{039}{241}{206}{071}{223}{251}{081}{018}{159}{173}{132}{209}{188}"

Message: you can't go in that direction

m$="000235160204003020000020"

This is the linearized representation of the [game map](https://github.com/spotlessmind1975/adv10en/blob/master/docs/game-state.md#game-map-and-rooms-connections). The map consists of six rooms, which are connected to each other by four directions. Each connection from one room to another is represented by a digit (0...6), which corresponds to the destination room, if you go along that certain direction (N, S, W, E). The room 0 mean "no way". Four digits make up all the connections starting from a room. There are a total of 24 digits.

d$(17)="{242}{078}{075}{238}{089}{044}{047}{175}{236}{080}{228}{044}{191}{028}{213}{235}{071}{239}{077}{172}{244}"

Message: a key, a rope, nothing more

d$(9)="{236}{086}{164}{031}{069}{239}{080}{148}{052}{033}{230}{044}{239}{089}{236}{085}{063}{068}{047}{239}{075}{228}{089}"

Message: over the pedestal, you see a key

d$(13)="{136}{239}{066}{238}{089}{045}{139}{062}{045}{242}{126}{051}{254}{048}{238}{066}{089}{239}{077}{055}{078}{080}{041}{070}{017}{253}"

Message: cc by-nc-sa 3.0 by m.spedaletti

d$(1)="{078}{087}{134}{236}{077}{244}{193}{159}{190}{085}{078}{071}{188}{247}{255}{081}{164}{228}{039}{243}{242}{201}{172}{047}{241}{036}{019}{247}"

Message: welcome to dungeon. there's a door at east.

3 d$(3)="{189}{047}{175}{204}{254}{077}{222}{087}{081}{047}{239}{080}{148}{052}{033}{246}{188}{079}{050}{225}{044}{047}{155}{047}{159}{029}{088}{047}{102}{047}{202}{190}{085}{121}"

Message: in a room with a pedestal on east, and a ditch all around.

d$(16)="{189}{047}{175}{204}{254}{077}{222}{087}{081}{047}{239}{080}{148}{052}{033}{118}{207}{078}{086}{250}{081}{244}{217}{129}{229}{044}{047}{175}{236}{080}{244}{061}{095}{178}{222}{071}{235}{071}{207}{251}{036}{019}{247}"

Message: in a room with a pedestal. over the ditch, a rope is hanging on east.

d$(2)="{189}{031}{069}{239}{077}{157}{105}{244}{236}{070}{047}{251}{210}{099}{116}{239}{089}{236}{085}{063}{068}{031}{206}{087}{175}{204}{062}{077}{047}{241}{203}{026}{245}{178}{249}{195}{030}{085}{117}{047}{159}{204}{250}{061}{047}{241}{036}{019}{247}"

Message: in the middle of an aisle. you see two rooms at north and south. a door is at east.

print "{light green}";:z=1:e$=d$(12):f=len(e$):gosub9:print

Ok, so this is the first time that we call [the decompression routine that prints a description](https://github.com/spotlessmind1975/adv10en/blob/master/docs/compression.md#a-readable-decoder) (line 9). We will call it several times. In **e$** we put the text to be printed, in **f** we put its length, while in **z** we put the position from which to start to decompress. Before printing, we set the color green (for titles).

So, print the first line of titles.

z=1:e$=d$(13):f=len(e$):gosub9:print:print

Second line of titles (and an empty line more):

## GAME LOOP (LINES 4-8)

From this point on, the program's lines are ordered according to a game logic.

### ROOM DESCRIPTION AND SCORE (LINES 4-5)

4 print "{light gray}";

This is where the game loop's starts. The player will return to this line (line 4) each time the description of the room is needed. The room description is print using light gray color.

|  |  |
| --- | --- |
| z = -  (x=0)-  (x=1)\*15-  (x>0) | it's like:  [ IF X=0 THEN Z=1]  [ IF X=1 THEN Z=15]  [ IF X>0 THEN Z=1] |

The starting position depends [on the room the player is in](https://github.com/spotlessmind1975/adv10en/blob/master/docs/game-state.md#the-current-room-x) (room number zero is a special case). All rooms have a description that starts from position 1, except room 1, which has the position starting from character 15. With this expedient, we are able to differentiate the first description, so as to characterize the beginning of the game.

|  |  |
| --- | --- |
| x = -  (x=0)+  x | it's like:  [ IF X=0 THEN X=1]  [ IF X<>0 THEN X=X] |

If we're in room zero, let's move to room one (otherwise, stay there).

|  |  |
| --- | --- |
| e$ = d$( -  (x=0)+  x+  (x=3)\*(g=1)\*13) | it's like:  [ IF X=0 THEN E$=D$(1) ]  [ IF X=3 AND G=1 THEN E$=D$(16) ]  [ IF X<>0 THEN E$=D$(X) ] |

This is the description to be printed on the screen, which depends both on the room number (**x**) and, if the player is in the room no. 3, is different [if the player has used the rope or not](https://github.com/spotlessmind1975/adv10en/blob/master/docs/game-state.md#introduction) (**g**).

f=len(e$):gosub9

Finally, print the description of the room.

poke w+4,32

Stop the sound. The sound is produced by the routine that prints the description, and it is used to give a feedback to the player.

|  |  |
| --- | --- |
| z =  (x=4)\*(b=0)\*12+1 | it's like:  [ IF X=4 AND B=0 THEN Z=13 ELSE Z=1 ] |

Now let's print a specific sentence for [the boundary conditions](https://github.com/spotlessmind1975/adv10en/blob/master/docs/game-state.md#introduction). The starting position depends on the room where the player is and the condition that [the key has not been taken](https://github.com/spotlessmind1975/adv10en/blob/master/docs/game-state.md#introduction) if the player is on room nr. 4.

|  |  |
| --- | --- |
| q=  (x=5)\*(a=0)\*8+  (x=4)\*(b=0)\*9+  (x=3)\*(b=0)\*9 :  q = q +  (x=2)\*(h=0)\*15 | it's like:  [ IF X=5 AND A=0 THEN Q=8 ]  [ IF X=4 AND B=0 THEN Q=9 ]  [ IF X=3 AND B=0 THEN Q=9 ]  it's like:  [ IF X=2 AND H=0 THEN Q=9 ] |

This complex expression is used to calculate the index of which sentence to use, based on the [boundary conditions](https://github.com/spotlessmind1975/adv10en/blob/master/docs/game-state.md#introduction) and based on the [current room](https://github.com/spotlessmind1975/adv10en/blob/master/docs/game-state.md#the-current-room-x) in which the player is located.

|  |  |
| --- | --- |
| e$ = d$( -  (q>0)\*q-  (q=0)\*14) | it's like:  [ IF Q>0 THEN E$=D$(Q) ]  [ IF Q=0 THEN E$=D$(14) ] |

Note we will print a standard message if none of the conditions are met.

f=len(e$):gosub9:print

Finally, print the description of boundary conditions.

5 pokek,t+a+b+g+h:poken,1:pokek+1,t-1:poken+1,1:pokek+2,t+4:poken+2,1

We print the score. We do this by going directly to the video screen, and in doing so we prevent the score from disappearing due to scrolling when we arrive at the end of the screen.

### PARSER (LINES 5-6)

input"{gray}>";p$

We wait for the player to type something (in gray).

p1$=mid$(p$+m$,1,1)

p6$=mid$(p$+m$,6,1)

p7$=mid$(p$+m$,7,1)

Pass the string to the [parser](https://github.com/spotlessmind1975/adv10en/blob/master/docs/parser.md). To try to understand what the player wrote, let's examine the 3 letters on three different positions (1, 6 and 7). Note that he/she could type LESS than six characters: for this reason, we will append the string that represent the game map, in order to have enough letters. This is an optimization, and has nothing to do with the logic of the game.

|  |  |
| --- | --- |
| v = -  (p1$="n")-  (p1$="s")\*2-  (p1$="w")\*3  v = v -  (p1$="e")\*4-  (p1$="t")\*5-  (p1$="u")\*6-  (p1$="i")\*7 | it's like:  [ IF P1$="N" THEN V=1 ]  [ IF P1$="S" THEN V=2 ]  [ IF P1$="W" THEN V=3 ]  it's like:  [ IF P1$="E" THEN V=4 ]  [ IF P1$="T" THEN V=5 ]  [ IF P1$="U" THEN V=6 ]  [ IF P1$="I" THEN V=7 ] |

We decode the verb.

|  |  |
| --- | --- |
| 6 o =  (v=5)\*((p7$="o")+  (p7$="e")\*2)  o = o +  (v=6)\*  ((p6$="o")+(p6$="e")\*2) | it's like:  [ IF V=5 and P7$="O" THEN O=1 ]  [ IF V=5 and P7$="E" THEN O=2 ]  it's like:  [ IF V=6 and P6$="O" THEN O=1 ]  [ IF V=6 and P6$="E" THEN O=2 ] |

Next, we decode the object (if any).

### GAME LOGIC (LINES 6-7)

|  |  |
| --- | --- |
| a = -  (a=1)+  ((x=5)\*(a=0)\*(v=5)\*(o=1)) | it's like:  [ IF A=1 THEN A=1 ]  [ IF X=5 AND A=0 AND V=5 AND O=1 THEN A=1 ] |

If action is "take rope" (v=5 and o=1), we will update the status, if the rope has not been taken already (a=1) and it is in the right room (x=5).

|  |  |
| --- | --- |
| b = -  (b=1)+  ((x=4)\*(b=0)\*(v=5)\*(o=2)) | it's like:  [ IF B=1 THEN B=1 ]  [ IF X=4 AND B=0 AND V=5 AND O=2 THEN B=1 ] |

If action is "take key" (v=5 and o=2), we will update the status, if the key has not been taken already (b=1) and it is in the right room (x=4).

|  |  |
| --- | --- |
| g = -  (g=1)-  ((a=1)\*(x=3)\*(g=0)\*(v=6)\*(o=1)) | it's like:  [ IF G=1 THEN G=1 ]  [ IF A=1 AND X=3 AND G=0 AND V=6 AND O=1 THEN G=1 ] |

If action is "use rope" (v=6 and o=1), we will update the status, if the rope has not been used already (g=1) and it is in the right room (x=3).

|  |  |
| --- | --- |
| h = -  (h=1)-  ((b=1)\*(x=2)\*(h=0)\*(v=6)\*(o=2)) | it's like:  [ IF H=1 THEN H=1 ]  [ IF B=1 AND X=2 AND H=0 AND V=6 AND O=2 THEN G=1 ] |

If action is "use key" (v=6 and o=2), we will update the status, if the key has not been used already (h=1) and it is in the right room (x=3).

|  |  |
| --- | --- |
| r=  (v>0)\*(v<5)\*val(  mid$(m$,(x-1)\*4-(v>0)\*v-(v=0),1)  ) | it's like:  [ IF V>0 AND V<5 THEN  R=VAL(MAPPA) ] |

Now let's move on to calculating what the next room would be IF the conditions were met. First, if the verb is one of the moving ones, the room should be the one indicated by the map.

|  |  |
| --- | --- |
| r=r -  x\*(v>4)-  x\*(v=0) | it's like:  [ IF V>4 THEN R=X ]  [ IF V=0 THEN R=X ] |

However, if the verb were not moving, the next room will be always the current one. The same happens if the program doesn't understand what the player means.

|  |  |
| --- | --- |
| r =  ((r=4)\*(g=1))\*4+  ((r=6)\*(h=1))\*6+  ((r<>4)\*(r<>6))\*r | it's like:  [ IF R=4 AND G=1 THEN R=4 ]  [ IF R=6 AND H=1 THEN R=6 ]  [ IF R<>6 AND R<>4 THEN R=R ] |

Whichever room the player will go to, the conditions necessary for moving must be met.

|  |  |
| --- | --- |
| x = -  (r>0)\*r-  x\*(r=0) | it's like:  [ IF R>0 THEN X=R ]  [ IF R=0 THEN X=X ] |

Ultimately, if the next room is zero (0), it means that the player must stay where he/she is; otherwise, it will move accordingly.

### RESPONSE (LINE 8)

Now we have to calculate the answer. It involves testing a number of possibilities:

* that the program did not understand what the player meant;
* that the player wanted to move, but he/she cannot;
* that the player wanted to do something, but he/she could not do it.

|  |  |
| --- | --- |
| j = -  (r=0)-  (v=0)-  ((v>4)\*(v<>7)\*(o=0))  j = j -  ((v=5)\*(o=1)\*(a=0))-  ((v=5)\*(o=2)\*(b=0))  8 j = j -  ((v=6)\*(o=1)\*(g=0))-  ((v=6)\*(o=2)\*(h=0)) | it's like:  [ IF R=0 THEN J=1 ]  [ IF V=0 THEN J=1 ]  [ IF V>4 AND V<>7 AND O=0 THEN J=1 ]  it's like:  [ IF V=5 AND O=1 AND A=0 THEN J=1 ]  [ IF V=5 AND O=2 AND B=0 THEN J=1 ]  it's like:  [ IF V=6 AND O=1 AND G=0 THEN J=1 ]  [ IF V=6 AND O=2 AND G=1 THEN J=1 ] |

For each of these conditions (**j=1**), there is a specific message. There is a specific message also for the condition that everything is ok (**j=0**) and the player can move to another room (or stay in the one where he/she is).

|  |  |
| --- | --- |
| e$ = d$( -  (j=1)\*(-(r=0)\*7-  (r>0)\*10)-  (j=0)\*(-(v=7)\*17-  (v<>7)\*11)) | it's like:  [ IF J=1 AND R=0 THEN E$=D$(7) ]  [ IF J=1 AND R>0 THEN E$=D$(10) ]  [ IF J=0 AND V=7 THEN E$=D$(17) ]  [ IF J=0 AND V<>7 THEN E$=D$(11) ] |

Again, there are differentiated messages, whether there is an error (**j=1**) or not (**j=0**).

|  |  |
| --- | --- |
| print  chr$(-(j=1)\*28-  (j=0)\*30) | it's like:  [ IF J=1 THEN PRINT CHR$(28) ]  [ IF J=0 THEN PRINT CHR$(30) ] |

The color of the writing denotes whether there was an error (red color) or not (green color). Green is PETSCII 30 while red is PETSCII 28.

|  |  |
| --- | --- |
| z = -  (j=1)+  (j=0)\*(v<7)+  (j=0)\*(v=7)\*((b=1)\*(a=1)+  (b=0)\*(a=1)\*7+  (b=0)\*(a=0)\*13) | it's like:  [ IF J=1 THEN Z=1 ]  [ IF J=0 AND V<7 THEN Z=1 ]  [ IF J=0 AND V=7 AND A=1 AND B=1 THEN Z=1 ]  [ IF J=0 AND V=7 AND A=1 THEN Z=7 ]  [ IF J=0 AND V=7 AND A=0 THEN Z=13 ] |

The starting point to write also depends on the message and the [boundary conditions](https://github.com/spotlessmind1975/adv10en/blob/master/docs/game-state.md#introduction). This is the logic that handles the response to be returned when inventory is requested (**v=7**). The algorithm causes the writing to be sent on the screen depending on the "possession" of objects. The entire message is written when the player owns every object, while partial message is printed if the player lacks of some objects.

f=len(e$):gosub9:print:print

Finally, print the response (and an empty line).

on j+1 goto 4,5

If the response was an error (**j=1**) then we move to the point where we ask for user input (line 5); otherwise (**j=0**) we repeat the game loop, which will propose a new room to the player (line 4).

9 w$=mid$(e$,z,1):y=asc(w$):

n0=yand15:n1=(y/16)and15

v0=(n0=14):v1=(n1=14):

l0=(n0<14):l1=(n1<14)

print

chr$(-v0\*asc(mid$(e$,(-v0\*z)+1,1))-

(n0=15)\*32-l0\*asc(mid$(v$,n0,1)));

chr$(-v1\*asc(mid$(e$,(-v1\*z)+2+l0+(n0=15),1))-

(n1=15)\*32-l1\*asc(mid$(v$,n1,1)));

z=z-v0-v1+1:ifz<=fthen9

This is the text decompression routine ([see here](https://github.com/spotlessmind1975/adv10en/blob/master/docs/compression.md#a-readable-decoder) for a more readable program).

10 pokew+5,9 : pokew+6,0

pokew+24,15 : pokew+1,25

pokew,177 : pokew+4,33

Although this area is still part of the decompression routine, we have placed some code and logic here, because there was no room elsewhere. In particular, we make a sound when we have finished printing a string.

if x<>6 or f<10 then return

If we find ourselves printing the last sentence of the game, then the game will continue running (ending). Otherwise, let's go back to the call point, to continue with the execution.

# DEDUCTIVE PARSER

The parser represents one of the greatest difficulties for those who program a text adventure game. Partly because the natural language is ambiguous, and it requires a lot of analysis to understand the intention of the player; a bit because the extreme freedom left to the player determines a great effort in predicting the most frequent interactions.

With few lines available, it was unthinkable to introduce a sophisticated parser. On the other hand, wanting to leave the illusion of a freedom of typing, it was still necessary to manage the most important cases, returning a message of misunderstanding in the negative. The little size of the screenplay made it possible to adopt a deductive input analysis mechanism. It was therefore possible to deduce the set of words to be recognized and their combination. In short, **the presence of certain letters in typing implies the interpretation of those specific commands and actions**.

Below we give the needed set of verbs:

|  |  |  |  |
| --- | --- | --- | --- |
| NORTH | SOUTH | WEST | EAST |
| TAKE | USE | INVENTORY |  |

Below we give the needed set of nouns:

|  |  |
| --- | --- |
| ROPE | KEY |

This is the combinational matrix of both tables, and it represent the entire set of commands that the parser has to recognize:

|  |  |  |  |
| --- | --- | --- | --- |
| NORTH | SOUTH | WEST | EAST |
| TAKE ROPE | TAKE KEY | USE ROPE | USE KEY |
| INVENTORY |  |  |  |

In order to distinguish each command, it will be sufficient to perform a sampling of what has been written, to identify what distinguishes one of these expressions from the other.

[](https://github.com/spotlessmind1975/adv10en/blob/master/docs/parser.png)

In the specific case, it was chosen to sample the character in position 1, 6 and 7. Based on this analysis, two status variables can be populated: the verb (**v**) and the object (**o**). These two numbers can therefore be manipulated more quickly and concisely, and they can be used to change the value of status variable to proceed with the game as detailed in the [internal game state](https://github.com/spotlessmind1975/adv10en/blob/master/docs/game-state.md).

Whether the parser understands or not the player's input, an appropriate message will be printed among those supplied (green if ok, red if is is an error). Also in this case, as in the case of descriptions and states, the choice of the message is made according to certain conditions (see [internal game state](https://github.com/spotlessmind1975/adv10en/blob/master/docs/game-state.md) for more information).

PS. to have a better user experience, a sound note will be emitted at each interaction: a sort of "beep", which serves to indicate that the system is ready to accept the next command.

# INTERNAL GAME STATE

\*\*SPOILER ALERT\*\*

By continuing to read, you will become aware of some details of the game, which could preclude a playful experience.

## INTRODUCTION

The state of the game is represented by a set of variables, which describe the salient facts of the game. The variables follow:

* **x**: in which room is the player located? 0-6
* **a**: did the player take the rope? 0 = no, 1 = yes
* **b**: did the player take the key? 0 = no, 1 = yes
* **g**: did the player use the rope in the room with the pedestal? 0 = no, 1 = yes
* **h**: did the player use the key in the aisle with the closed door? 0 = no, 1 = yes

## THE CURRENT ROOM (x)

[](https://github.com/spotlessmind1975/adv10en/blob/master/docs/map.png)

The room where the player is located (**x**) is one of the six rooms of the game, arranged as follows (see map):

* **0-1**: beginning room
* **2**: aisle
* **3**: pedestal room
* **4**: over the pedestal (room center)
* **5**: torture chamber
* **6**: exit!

The repetition of room "beginning room" for values 0 and 1 is needed to make a different description of the room. The difference is between the first time it is displayed to the player and the next one, and it lies entirely in the starting point of description decoding. This technique allows to add and additional level of compression on descriptions. It is used also in other parts of the game (such as, for example, in the inventory). So, the first time the description of the initial room is displayed, the room number is automatically changed from 0 to 1 (**x = 1**).

## GAME MAP AND ROOM CONNECTIONS

[](https://github.com/spotlessmind1975/adv10en/blob/master/docs/directions.png)

The connections between rooms are described by a single matrix of 6 rows by 4 columns, linearized as a string in the variable **m$**. A row of 4 digits represents a room, while each column (6 digits) represents the new value of **x** if the user chooses to move (respectively) **N**orth (first digit), **S**outh (second digit), **W**est (third digit) or **E**ast (fourth digit) on each room.

[](https://github.com/spotlessmind1975/adv10en/blob/master/docs/directions2.png)

So if the player is in room no. 2 (**x = 2**) and he enters N, the game will take him to room no. 3 (**x = 3**).

Although in the current game all connections are bidirectional, this is not mandatory. It would have been possible to represent, for example, "sliding" into a room from which the player could not exit, thus forcing the player in the path of adventure.

## THE STATUS VARIABLES (a,b,g,h)

We present a different description according to the room in which the player is (see above), and the current status of the other status variables (**a,b,g,h**). To ensure the best possible description of the situation, the state of the game is represented by two related descriptions:

* **room description**: the situation of the room and describes it in a generic way (driven by the variable **x**);
* **status description**: it represents any additional elements that can change according to the state of the variables (driven by the variable **q**, that is a linear combination of status variables)

Since both descriptions are always present, if there is no status description (**q = 0**) this is represented by a standard message (NOTHING MORE).

## STATUS BASED MOVEMENT CONSTRAINTS

In addition to explicit movement constraints, such as those induced by the matrix (see above), there are also constraints that are based on the current state variables (orange coloured background).**Those costraints prevent the player from moving to some rooms until certain conditions are met.** This happens, for example, in the aisle room, when a locked door block the player's exit.

State changes intersect with the [parser](https://github.com/spotlessmind1975/adv10en/blob/master/docs/parser.md). Infact, once the verb (**v**) and noun (**n**) have been decoded, their values are used to vary the state of the aforementioned variables. This variation is subject to feasibility criteria, that is (i.e.) to being in the right room or having previously taken possession of other needed objects. Once calculated, the destination room is calculated and the parser will check again the complesive result. If something goes wrong, it will emit an error.

[](https://github.com/spotlessmind1975/adv10en/blob/master/docs/map_rope.png)

To better understand the mechanism, take the example of the rope object. This object (**o = 1**) is located in room no. 5 (**x = 5**), and it is described on that room when the player enters. If the parser understands that the player has requested to take (**v = 5**) just that object, the game will check if this is possible, with this formula: the rope can only be taken only and only if it has not already been taken (**a = 0**) and the player is in the room. nr. 5 (**x = 5**).

In short:

a=-(a=1)+((x=5)\*(a=0)\*(v=5)\*(o=1))

In the absence of one of those conditions, **the state does not change at all** and, in the absence of possession condition (**a = 1**) when user asked to take (**v = 5**) the rope (**a = 1**), the parser will emit an error. By using the current status variables, the program will block the movement by calculating a 0 for the destination room, if conditions are not met. Again, the parser will emit an error if next room is zero (**x = 0**) when a verb is a movement (**v < 5**).

## THE SCORE

The progress of the game, in terms of interaction with the environment, is represented by a score. **The score depends on the sum of all the states indicated above**. The score, which is drawn in the upper right part of the screen and is updated with each interaction, is declined as follows:

* score at the beginning: **0/4**
* when you take the rope: **1/4**
* when you use the rope (in the right place): **2/4**
* when you take the key: **3/4**
* when you use the key (in the right place): **4/4**

# NIBBLE TEXT COMPRESSION

One of the most "bulky" elements of the programs written in BASIC V2 are certainly the texts: error messages, descriptions, labels, prompts ... everything must be strongly reduced in order to save precious space for the rest of the program. This applies to both more and less serious applications, such as games, and it becomes critical in those games where texts are essential.

I am referring, of course, to the so-called "text adventures" and it would be fantastic if we found a way to represent texts in a more "compact" way.

The best way to solve this problem in general is to take into consideration the low variability of the letters and, therefore, to use a nibble representation (4 bit) instead of a byte (8 bit).

[](https://github.com/spotlessmind1975/adv10en/blob/master/docs/nibble.png)

This approach would ideally reduce the occupation of space to exactly half of what is needed today. Obviously, we can represent only 16 symbols (0...15), which are insufficient to completely represent all the letters of the alphabet, and one punctuation mark (28 letters: a-z, the full stop "." and the space).

But, if we assigned the letters that are most frequent in that text to these 16 symbols, **we would statistically reduce the occupation** since we will use more frequently a nibble instead of a byte.

In order to be able to encode any generic text, we must sacrifice some of the representable values to introduce a way to return (temporarily) to the 8-bit representation, when we need it.

To give an example, I will take the following text from [10LINES ADV (English edition)](https://github.com/spotlessmind1975/adv10en):

IN THE MIDDLE OF AN AISLE. YOU SEE TWO ROOMS AT NORTH AND SOUTH. A DOOR IS AT EAST.

This string is exactly 83 characters long and it has 17 different letters. If we represent it using 8 bit chars, it would occupy exactly 83 bytes.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| "I" | "N" | (spazio) | "T" | "H" | "E" | ... |

Now, we suppose to use the following correspondence table. I built this table by ordering the letters from the most frequent to the least one in that text:

|  |  |  |
| --- | --- | --- |
| **LETTER** | **DECIMAL (4 bit)** | **HEXADECIMAL (4 bit)** |
| "o" | 1 | 1 |
| "t" | 2 | 2 |
| "a" | 3 | 3 |
| "s" | 4 | 4 |
| "e" | 5 | 5 |
| "n" | 6 | 6 |
| "i" | 7 | 7 |
| "d" | 8 | 8 |
| "h" | 9 | 9 |
| "." | 10 | A |
| "r" | 11 | B |
| "l" | 12 | C |
| "m" | 13 | D |
| (escape) | 14 | E |
| (spazio) | 15 | F |

Now let's take up the previous text and express it by using the hexadecimal numbers from the previous frequency table:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| "I" | "N" | (spazio) | "T" | "H" | "E" | ... |
| 7 | 6 | F | 2 | 9 | 5 | ... |

Since all these numbers can be represented with only one nibble (4 bit), they can be "grouped" in a single byte each two nibbles (8 bit):

|  |  |  |  |
| --- | --- | --- | --- |
| "IN" | (spazio) "T" | "HE" | ... |
| 76 | F2 | 95 | ... |

The text "in the", which previously occupied 6 bytes, after the "process" occupies only 3 bytes. **We therefore saved 50% of the space!**

If the letter is not among those of the correspondence, we need of an additional value, the (escape) value (14, hex. E). This value represents an "escape sequence", and it indicates that the letter will not be represented with a nibble but with the subsequent byte.

In the example above, if we had to translate "of" we would have had the following sequence:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| "O" | "F" |  |  | ... |
| 1 | E | 4 | F | ... |

So: 2 bytes for 2 bytes, and obviously there is no savings.

It can happen that there are two letters that cannot be translated into a sequence. In this case, we will have **TWO** escape sequences and two characters in the queue: therefore 3 bytes for 2 bytes, and we will have a loss.

I.e.:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| "F" | "F" |  |  |  |  | ... |
| E | E | 4 | F | 4 | F | ... |

Unfortunately, depending on the destination system where the byte sequence thus obtained will be represented, some of these combinations may not be valid, for one or more of these reasons (they are sorted from the most serious to the lightest one, and the level between brackets is about "severity"):

* **the correct listing of the program could be prevented**, because the characters would appear several times or would not appear at all, when you give the LIST command, or would generate a syntax error (LEVEL 1);
* **you could prevent typing the listing using the integrated editor**, because some characters would not be typeable (LEVEL 2);
* **"re-editing" could be prevented**, because the representation could change with a subsequent modification (LEVEL 3);
* **ambiguous characters could be introduced**, for which there are several corresponding codes (LEVEL 4).

With this in mind, I conducted an empirical study on the characters of the COMMODORE 64, which led me to draw up an algorithm ranking according to the aforementioned level of severity in their side effects, if they were adopted:

|  |  |
| --- | --- |
| **LEVEL** | **CHARS (dec./hex.)** |
| LEVEL 1 | |  |  |  | | --- | --- | --- | | 0 | 00 |  | | 10 | 0A | LINE FEED. | | 13 | 0D | ENTER. | | 20 | 14 | It is the character used to represent the delete of one character. | | 34 | 22 | Quotes are not allowed. | | 141 | 8D |  | |
| LEVEL 2 | |  |  |  | | --- | --- | --- | | 130 | 81 | These are characters for which there is no sequence on the C=64 keyboard capable of introducing them. | | 131 | 82 | | 132 | 83 | | 143 | 8f | | 222 | de | |
| LEVEL 3 | |  |  |  | | --- | --- | --- | | 96 | 60 | Automatic conversion. | | ... | ... | | 127 | 7f | | 224 | e0 | Automatic conversion. | | ... | ... | | 253 | 7f | |
| LEVEL 4 | |  |  |  | | --- | --- | --- | | 192 | c0 | Not unique code for same character. | | ... | ... | | 223 | df | | 160 | a0 | Not unique code for same character. | | ... | ... | | 190 | 254 | |

Although the double escape is always available (replacing the offending byte with a double nibble "EE" followed by the two characters "as is") this strategy is not always successful or necessary. For example, if you limit yourself to a compressor of LEVEL 1 (the lowest level of compatibility), it would be enough to escape only those few characters.

On the other hand, the order in which the letters are put in the dictionary can also make a difference: in fact, these two dictionary sequences

[](https://github.com/spotlessmind1975/adv10en/blob/master/docs/letters.png)

they produce **exactly** the same level of compression: however, they organize the nibbles, and therefore the bytes, in a different way. It is thus possible to imagine finding **a different order** of these letters, in order to minimize the use of escape sequences.

Finally, in order to evaluate how much saving we could reach in applying this compression schema, we must verify the relationship between the space occupied by all the elements needed to reconstruct the text with respect to the length of the original text. Obviously, being dependent on the content of the text, **it is not possible to give a deterministic measure of the space reduction ratio**. We can, however, present a (pessimistic) estimate model that takes into account the following conditions:

* **that 50% of the text** can be encoded by a sequence of 2 nibble (2 bytes to 1 bytes);
* **that 40% of the text** can be encoded by a sequence of 1 nibble ad an escape sequence (2 bytes to 2 bytes);
* **that 10% of the text** can be encoded by a sequence of two escape sequence (2 bytes to 3 bytes).

[](https://github.com/spotlessmind1975/adv10en/blob/master/docs/compression_ratio.png)

This graph shows the compression ratio trend in function of the input text, with the rules set out above about the content.

The trend of the graph with the various lengths presents a balance point around 70-90 characters, and it improves markedly with increasing length of the texts (x axis is in logarithmic scale).

For example, with a text of 256 characters, a saving of 46 characters is obtained, so reaching a compress ratio of about 86%. With 1 KB of text you get a ratio of 81% and a net saving of 190 characters. In reality, **the savings are greater and that makes the use of this technique very advantageous**, especially if texts can be modified to optimize the content.

## A READABLE DECODER

In order to make this algorithm reusable, the decoder has been implemented **as a single line of BASIC V2 code** of exactly 247 characters. This routine is located on line 9 of the **adv10en** program and it can be called as subroutine (GOSUB 9), as long as you use RETURN in the following lines.

You must pass:

* in **v$** the letter's dictionary (a string of 16 characters);
* in **e$** the byte sequence to decompress;
* in **f** the length of the byte sequence;
* in **z** the starting char to be decompressed.

The commented routine follows, where the "in one line" techniques have been abandoned in favor of better readability:

50 REM \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

51 REM \*\*\* NIBBLE DECOMPRESSOR (BASIC V2) by m.spedaletti (asimov@mclink.it)

52 REM \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

53 REM \*\*\* UTILIZZO:

54 REM \*\*\* v$ = dictionary (16 symbols, used only first 13)

55 REM \*\*\* e$ = byte sequence to decode

56 REM \*\*\* f = length of sequence to decode

57 REM \*\*\* z = first position to decode

58 REM \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

59 REM

60 REM ESEMPIO (tratto da "adv10en.bas")

61 v$="eoatnsidrh.lcumy"

62 e$="{087}{079}{026}{239}{077}{135}{200}{241}{226}{070}{063}{245}{115}{198}{177}{239}{089}{226}{085}{111}{017}{079}{046}{087}{159}{238}{079}{079}{110}{077}{063}{244}{037}{073}{250}{083}{248}{038}{078}{085}{186}{063}{143}{238}{079}{079}{249}{103}{063}{244}{049}{070}{235}{032}"

63 f=len(e$)

64 z=1

65 GOSUB 100

66 END

100 REM NIBBLE DECOMPRESSOR

110 REM Take z-nth character to decode

120 w$=mid$(e$,z,1)+e$

130 REM Convert it into a byte

140 y=asc(w$)

150 REM Extract two nibbles

160 n0=yand15:n1=(y/16)and15

170 REM Are they a space?

180 s0=(n0=15):s1=(n1=15)

190 REM Are they an escape sequence?

200 v0=(n0=14):v1=(n1=14)

210 REM Are they a letter?

220 l0=(n0<14):l1=(n1<14)

230 REM Print first and second nibble

240 if v0 then print mid$(e$,z+1,1); REM first = escape

250 if s0 then print " "; REM first = space

260 if l0 then print mid$(v$,n0,1); REM first = letter

270 if v1 then print mid$(e$,z+(v0=0)+2,1); REM second = escape

280 if s1 then print " "; REM second = space

290 if l1 then print mid$(v$,n1,1); REM second = letter

300 REM Go on on sequence, taking care that each escape

310 REM sequence means to go on by one additional byte.

320 REM So we could go ahead from 1 to 3 bytes.

330 z=z-v0-v1+1

340 REM Is sequence decoding finished?

350 ifz<=fthen100

360 return