Choose Your Destiny Tutorial

- Choose Your Destiny Tutorial
  - Introduction
  - Installation
    - Windows
    - Linux, BSDs, etc. (Experimental)
    - Highlighter for VSCode
  - Preparing our first adventure
  - Formato del fichero fuente
  - Jumps and labels
  - Options
  - Pauses and Waits
  - Text layout on screen
  - Images
  - Sound effects (Beeper)
  - Versions for the different Zx Spectrum models
  - Music (AY)
  - Variables
  - Variable declaration
  - Subroutines
  - Conditional Execution
  - Loops
  - Text compression and abbreviations
  - Workflow
  - Side-scrolling menus
  - Copying image fragments to screen
  - Reading the keyboard, variable arrays and indirections
    - Indirection
    - Reading the keyboard (INKEY)
    - Deleting characters and making line breaks (BACKSPACE and NEWLINE)
  - Using the alternative character set
  - Windows
  - Arrays or sequences
  - Changing the character set

Introduction

ChooseYourDestiny (or **CYD** for short) is an environment that will allow you to create the gamebook experience on your Spectrum. The tool consists of a compiler that, using a simple language on a ready-made text, allows you to add interactivity and visual and sound effects to "play" this type of adventure.

When designing this type of tool there are different approaches. You can design a simple tool, easy to use and understand for anyone, but at the cost of reducing its flexibility. Or you can take a compiler of a general language for Spectrum such as **ZxBasic** from Boriel or **Z88DK**, where you already have to have advanced concepts of the machine and get into it, if necessary, even with assembler. For **CYD** I decided to take a middle ground approach: a simple, markup-based programming language, yet flexible enough to be able to create almost anything you want, but simplified enough to focus on the creative aspects rather than the technical ones.

Therefore, this tutorial has been created to introduce concepts that will allow you to understand how it works and start making your own adventures. This document is constantly being revised and updated but may be a bit out of date with respect to the reference distribution, so the absolute reference for information will always be the manual, which is always updated with the tool. I recommend that you always have it at hand.

In addition, there are also examples in the distribution's `examples` folder, which you can play with and learn from, and which will surely give you many ideas for your own creations. To try them out, simply copy the contents of each example folder and copy it to the root directory of the distribution, overwriting the files. If you have something already done that you want to keep, **remember to copy it somewhere else first**.

With this, I hope it helps you create adventures that we can all enjoy.

Installation

Windows

To install on Windows 10 (64-bit) or higher, download the `ChooseYourDestiny.Win_x64.zip` file from the Releases section of the repository and unzip it into a folder called Tutorial, which you can create wherever you see fit. The script to build the adventure is called `make_adv.cmd`, you will need to run it to compile the adventure. I recommend doing this from the command line.

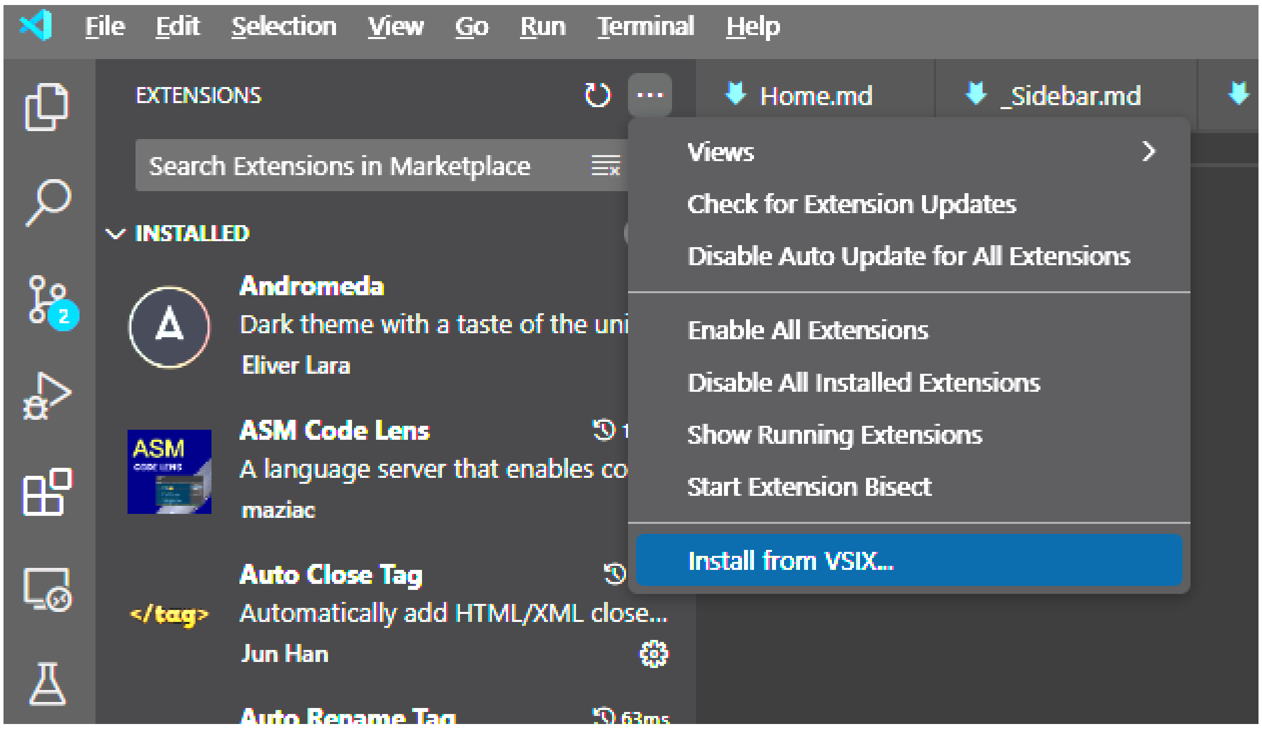
Linux, BSDs, etc. (Experimental)

You must first meet the prerequisites, for this see the relevant section of the manual.

Then download the `ChooseYourDestiny.Linux_x64.zip` file from the Releases section of the repository and unzip it into a folder called Tutorial, which you can create wherever you see fit. The script to build the adventures is called `make_adv.cmd`, you will need to run it to compile the adventure. I recommend doing this from the command line.

Highlighter for VSCode

To start using the extension with Visual Studio Code, download the file chooseyourdestiny-highlighter-x.x.x.vsix from Releases. On VsCode, go to the extensions screen, and on the ... button, it will open a new menu. From that, select the Install from VSIX option and open the previous file.


File Edit Selection View Go Run Terminal Help
EXTENSIONS
Home.md
_Sidebar.md
Search Extensions in Marketplace
INSTALLED
Andromeda
Dark theme with a taste of the uni
Eliver Lara
ASM Code Lens
A language server that enables co
maziac
Auto Close Tag
Automatically add HTML/XML close...
Jun Han
Auto Rename Tag
Views
Check for Extension Updates
Disable Auto Update for All Extensions
Enable All Extensions
Disable All Installed Extensions
Show Running Extensions
Start Extension Bisect
Install from VSIX...

Otherwise, download this repository and copy the folder into the `<user home>/.vscode/extensions` folder and restart Code. If your Code installation is portable, it must be copied on the folder `data/extensions/` inside of the VSCode folder.

Preparing our first adventure

The first thing we are going to do is change a couple of things so that we can generate a custom adventure.

If you are using Windows, open the file `make_adv.cmd` with any text editor and you will see this at the beginning:

```
REM ---- Configuration variables ----------

REM Name of the game
SET GAME=test
REM This name will be used as:
REM   - The file to compile will be test.cyd with this example
REM   - The name of the TAP file or +3 disk image

REM Target for the compiler (48k, 128k for TAP, plus3 for DSK)
SET TARGET=48k

REM Number of lines used on SCR files at compressing
SET IMGLINES=192

REM Loading screen
SET LOAD_SCR="LOAD.scr"

REM Parameters for compiler
SET CYDC_EXTRA_PARAMS=
REM ------
```

The first thing we are going to do is give a name to the adventure we are going to create. For example, we will call it `Tutorial`:

```
REM ---- Configuration variables ----------

REM Name of the game
SET GAME=Tutorial
REM This name will be used as:
REM   - The file to compile will be test.cyd with this example
REM   - The name of the TAP file or +3 disk image

REM Target for the compiler (48k, 128k for TAP, plus3 for DSK)
SET TARGET=48k

REM Number of lines used on SCR files at compressing
SET IMGLINES=192

REM Loading screen
SET LOAD_SCR="LOAD.scr"

REM Parameters for compiler
SET CYDC_EXTRA_PARAMS=

REM ------
```

If you use another operating system, the process is similar with `make_adv.sh`:

```
# ---- Configuration variables ----------
# Name of the game
GAME="Tutorial"
# This name will be used as:
#   - The file to compile will be test.cyd with this example
#   - The name of the TAP file or +3 disk image
#
# Target for the compiler (48k, 128k for TAP, plus3 for DSK)
TARGET="48k"
#
# Number of lines used on SCR files at compressing
IMGLINES="192"
#
# Loading screen
LOAD_SCR="./LOAD.scr"
#
# Parameters for compiler
CYDC_EXTRA_PARAMS=
# ----------------------------------------
```

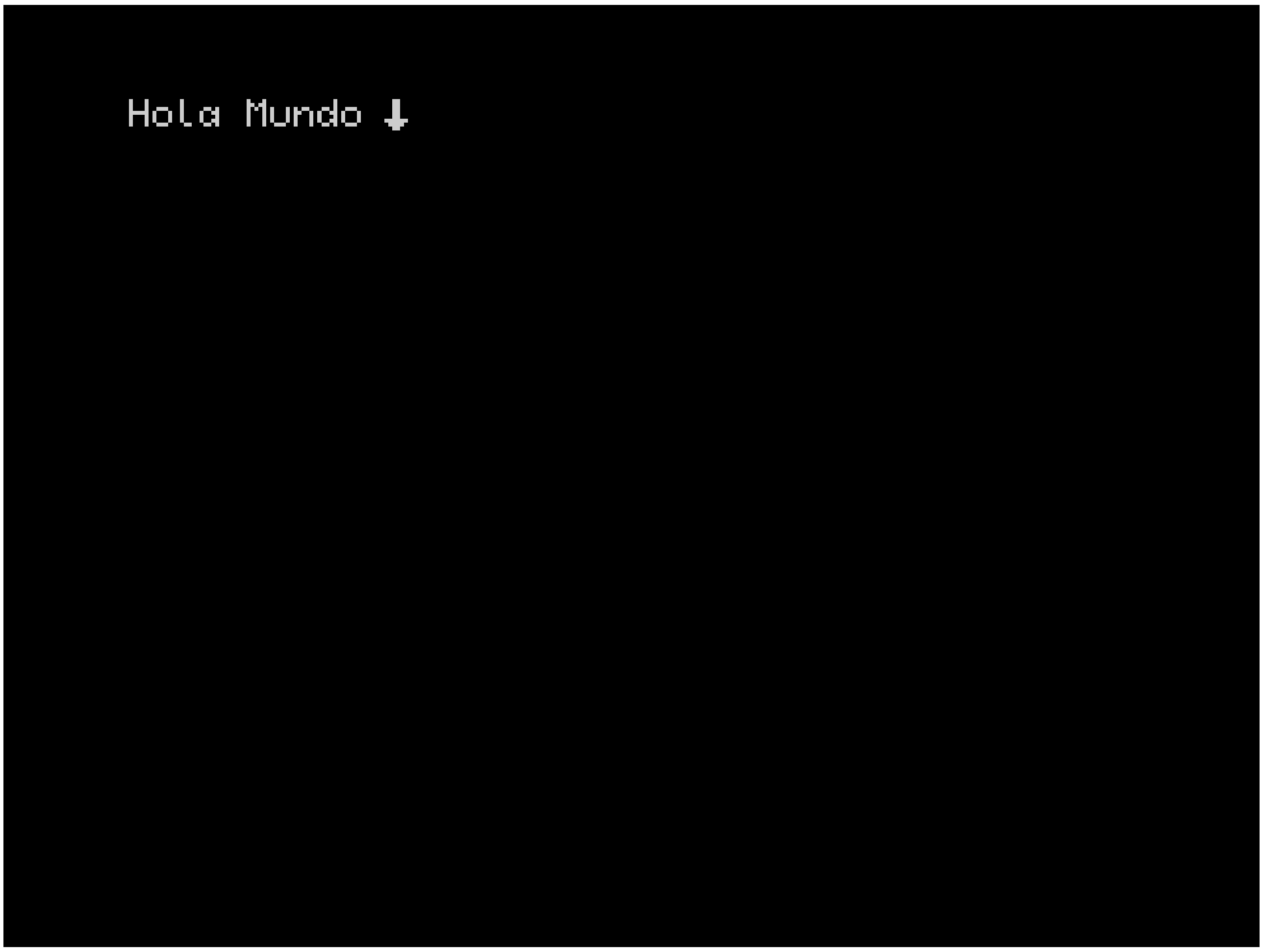
We save the file and now create a new text file, called `tutorial.Cyd`.In this file, we write this:

```
Hola Mundo[[WAITKEY]]
```

And we keep it in the same place as `Make_adv.cmd` and `make_adv.sh`. We execute the batch file and if everything goes well, you will have created a tape file called tutorial.TAP, which you can run with your favorite emulator.

Formato del fichero fuente

When launching the resulting Tap file with an emulator, this comes out:


```
Hola Mundo ⬇
```

Let's analyze what happens...

Apart from `GAME`, another important variable is `TARGET`, which indicates the Spectrum model and the type of output file to use. For now we will use the value `48k`, which we will later change when we want more advanced features.

Returning to the code of the adventure, we see that the text *Hello World* is painted and then a kind of cursor appears. If we press the `Enter` or `Space` key, the program is restarted. If we return to the code:

```
Hello World[[WAITKEY]]
```

There are two different parts, one is the *Hello World*, and then *[[WAITKEY]]*. The second part is a command that is sent to the interpreter to bring up that animated cursor and wait for a key to be pressed. This is the fundamental basis for understanding how the compiler works: **Everything between `[[` and `]]` is considered code or commands for the engine and everything outside is considered printable text**.

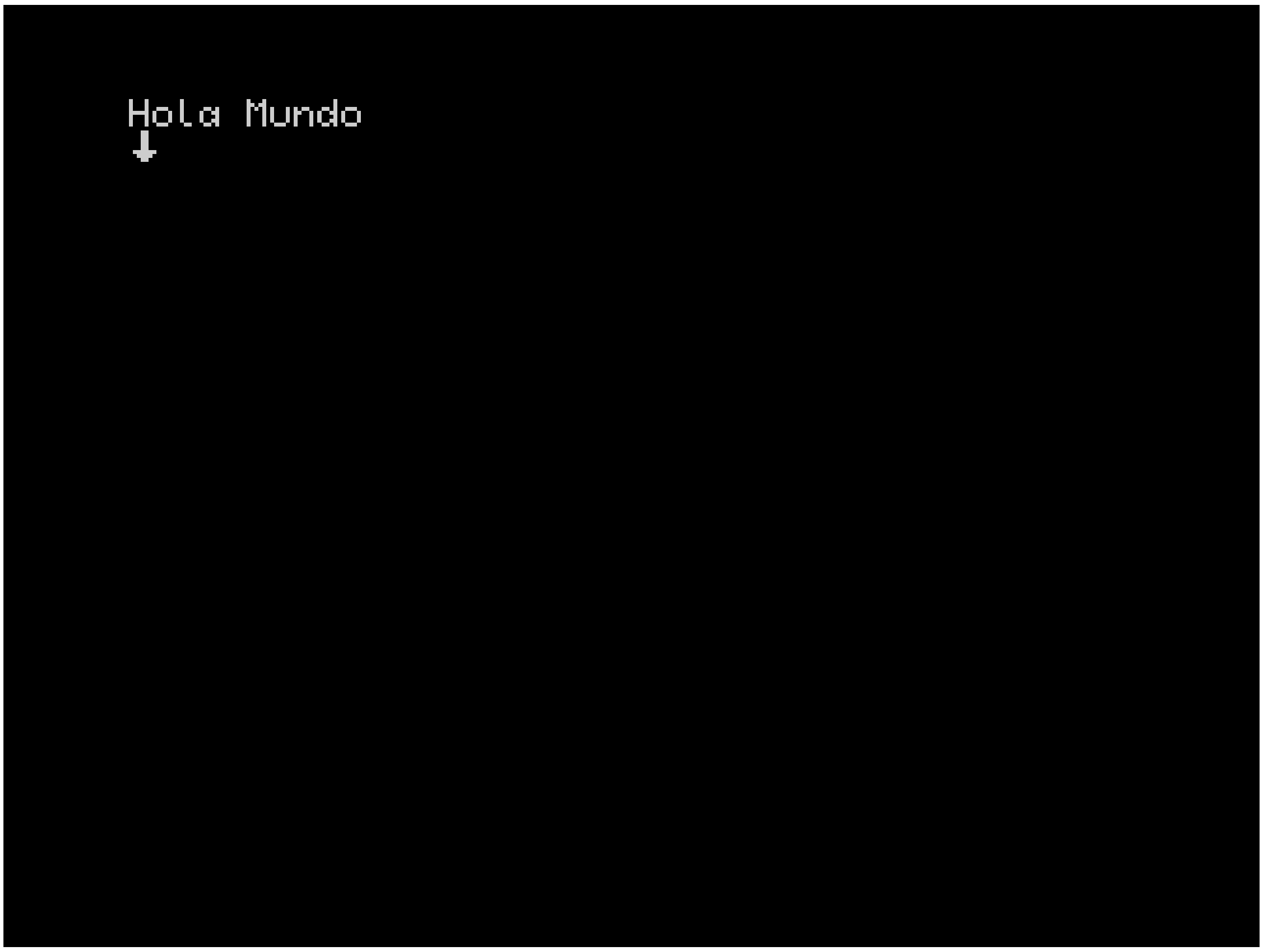
To understand this better, let's do an experiment. Let's put a line break after the two open brackets, like this:

```
Hola Mundo[[
WAITKEY]]
```

If we compile and load the game, we see that the same thing happens. Now, we delete the line break we put and put it **before** the double brackets:

```
Hola Mundo
[[WAITKEY]]
```

If we compile and load again, we see that now the icon is in the following line:

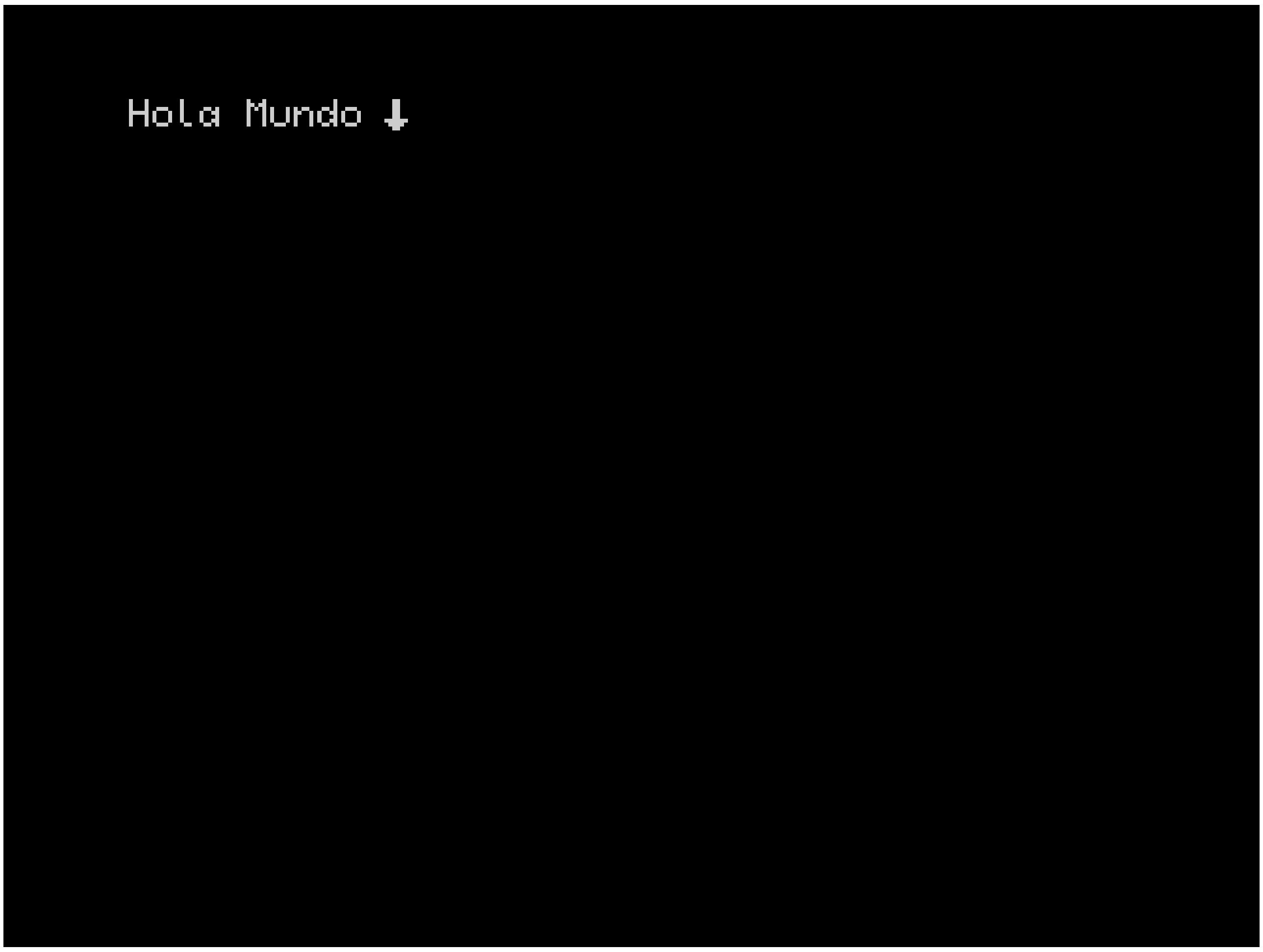

```
Hola Mundo
```

That is because the line jump, being out of the double brackets, is considered printable text and, therefore, the waiting icon passes to the next line. Keep these situations when you write the adventure.

Let's leave this as I was and we will add more commands. Type this within `tutorial.CyD`:

```
[[CLEAR]]Hola Mundo[[WAITKEY]]
```

Now we have put a command ahead of the text. If we compile and execute, we get this:

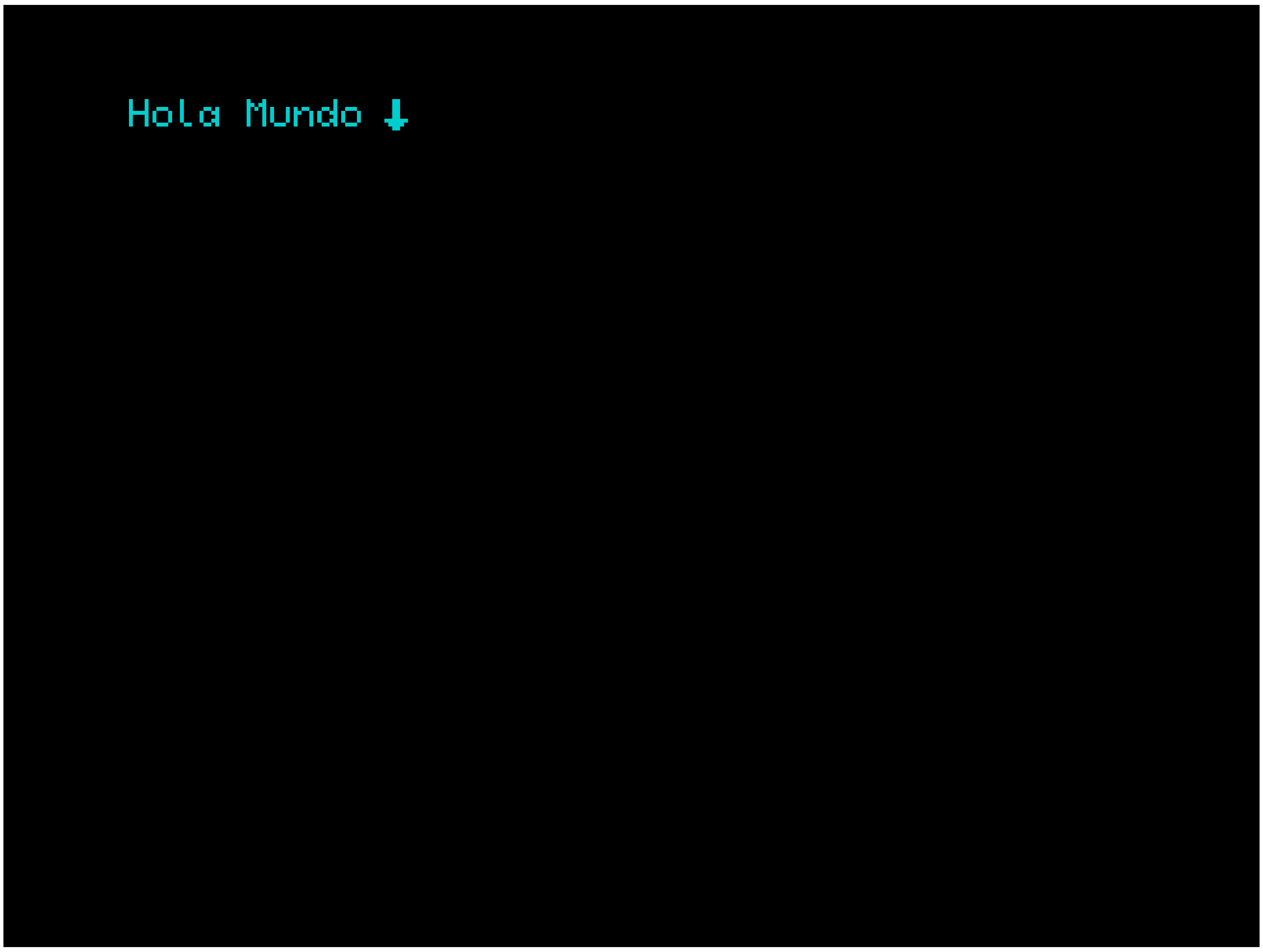

Hola Mundo ⬇

With the CLEAR command we delete the printable area that, for the moment, is the full screen. As the screen is automatically deleted at the beginning of the interpreter, we will not see anything at the moment, but with this command we will have the clean screen to print from the beginning.You have a complete reference of the commands in the [Manual] (https://github.com/cronomantic/ChooseYourDestiny/blob/main/manual_es.md).

Now we are going to change the color of the text.For this we are going to use the INK N command, where n is a number from 0 to 7 that corresponds to the colors of the spectrum.By default it is white, so we are going to put it with a cyan color, which is number 5, which would be ink 5. We will put it before Clear, such as: like this:

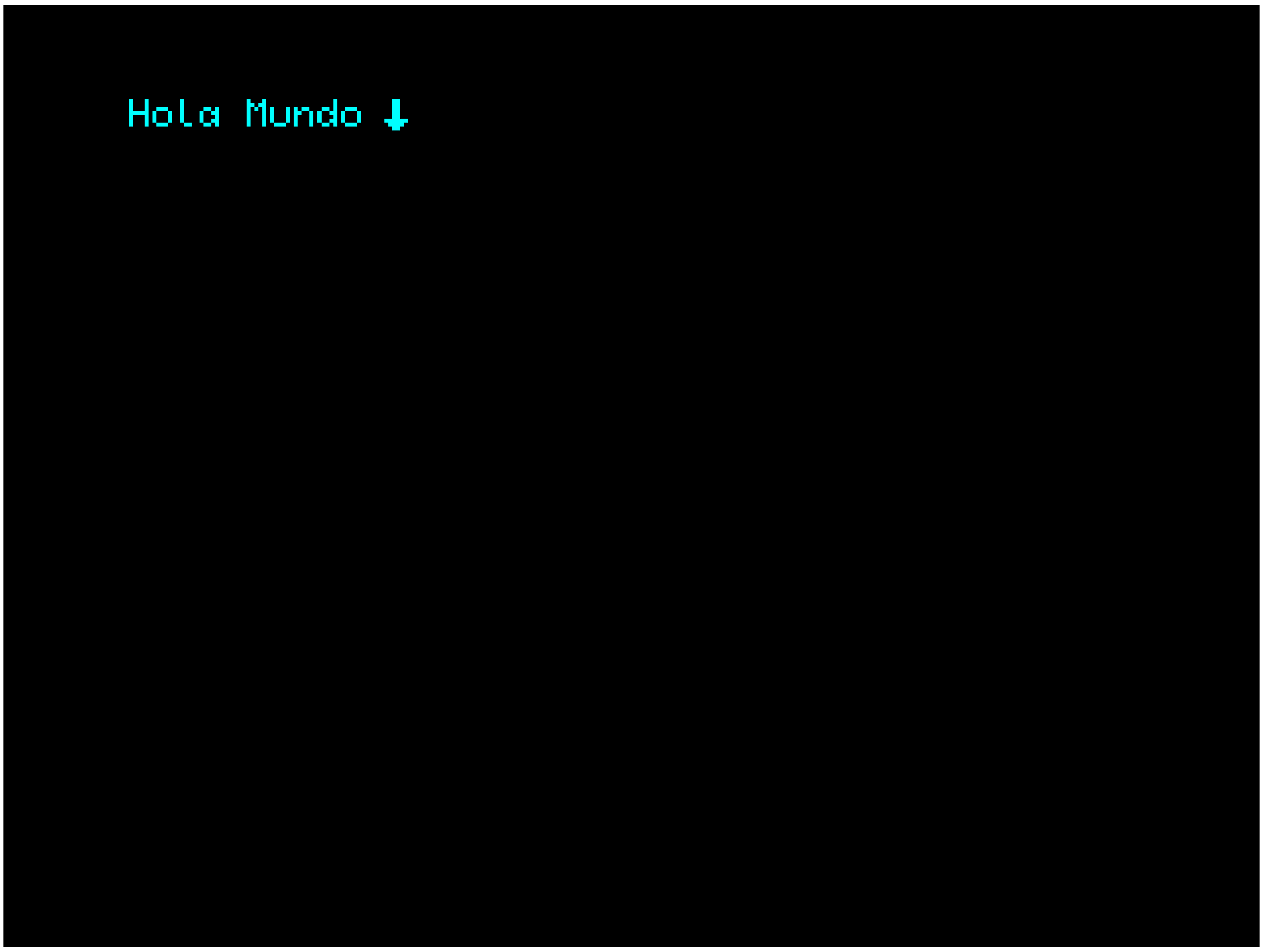
```
[[INK 5]][[CLEAR]]Hola Mundo[[WAITKEY]]
```

The result is ...


Hola Mundo ⬇

But the color is a bit dark ... let's bright it! To do this we use the Bright 1 command, (again, look at the reference in the manual), this way:

```
[[INK 5]][[BRIGHT 1]][[CLEAR]]Hola Mundo[[WAITKEY]]
```


Hola Mundo ⬇

This is already better.

But both bracket can be quite confusing and unpleasant to view. As I have already indicated, every time it is `[[`, the compiler interprets that the following are commands. How can we chain them without being opening and closing brackets? Well... there are two ways:

- Through line jumps:

```
[[
    INK 5
    BRIGHT 1
    CLEAR
]]Hola Mundo[[WAITKEY]]
```

- Through two points on the same line:

```
[[ INK 5 : BRIGHT 1 : CLEAR ]]Hola Mundo[[ WAITKEY ]]
```

The two previous variants will produce the same result and are equivalent.

One last point is the comments. Within the code we can put comments by surrounding them with `/*` and `*/`:

```
[[
    INK 5     /* Imprime en color Cyan */
    BRIGHT 1  /* Activamos brillo */
    CLEAR     /* Borramos la pantalla */
]]Hola Mundo[[
  WAITKEY     /* Espera a pulsar tecla */
]]
```

With this you should have a good notion of how the CyD source code works.

Jumps and labels

In the example from the previous chapter, you may have noticed that when we press the select key, the Spectrum is reset. This is because when we press the validation key (while waiting with WAITKEY), it reaches the end of the file. When this happens, the Spectrum is reset. We can do the same thing by using the END command anywhere in the code.

But we don't want it to do that. We want it to start over again. To do this, copy the following into the source file:

```
[[
  LABEL principio
  INK 5
  BRIGHT 1
  CLEAR
]]Hola Mundo[[
  WAITKEY
  GOTO principio
]]
```

When you compile and run it, it won't look very nice, but you'll notice that when you press the validate key, it doesn't reset, but rather clears the screen and prints the text and waits again.

The two additions to the code are `LABEL beginning` and `GOTO beginning`. The first command is not really a command, but a label, which puts a marker at that point with an identifier named *beginning*; and the second tells the interpreter to jump to where the label *beginning* is located.

The result is that, when you press the validate key with WAITKEY, the `GOTO beginning` is found and jumps to where the label *beginning* is declared, which, being the beginning, what it does is re-execute all the subsequent commands and print the text *Hello World*, and wait with WAITKEY again... In short, we have made an infinite loop.

However, we can improve the example like this:

```
[[
  INK 5
  BRIGHT 1
  LABEL principio
  CLEAR
]]Hola Mundo[[
  WAITKEY
  GOTO principio
]]
```

Now the label is declared just before the screen is cleared, and it will go there when the GOTO is reached, leaving the INK and BRIGHT unexecuted. Why? Because it is no longer necessary to execute them again, we have already set the text color and brightness at the beginning and executing them again is redundant!

The concept of labels and jumps is fundamental to understanding how to make a "Choose Your Own Adventure", since we will present options to the player, and depending on those options, we will go from one place to another in the text.

An important detail is the format of the label identifiers. These can only be a **sequence of numbers and letters or the underscore character in a row, and must start with a letter.**. That is, `LABEL 1` or `LABEL La Etiqueta` are not valid, but `LABEL l1` or `LABEL LaEtiqueta` are. They are also case sensitive, meaning that they are **case sensitive**, so `LABEL Etiqueta` and `LABEL Etiqueta` are not the same label. And obviously, you cannot declare a label with the same name twice.

On the other hand, commands are not case sensitive, meaning that `CLEAR`, `clear` or `Clear` are perfectly valid. However, I recommend capitalizing them to better distinguish them.

Starting with version 0.5, a shorthand way of declaring labels has been added, prefixing the label name with the character `#`, meaning that `LABEL Etiqueta` can be written as `#Etiqueta`. Thus, the previous example can be written like this:

```
[[
  INK 5
  BRIGHT 1
  #principio
  CLEAR
]]Hola Mundo[[
  WAITKEY
  GOTO principio
]]
```

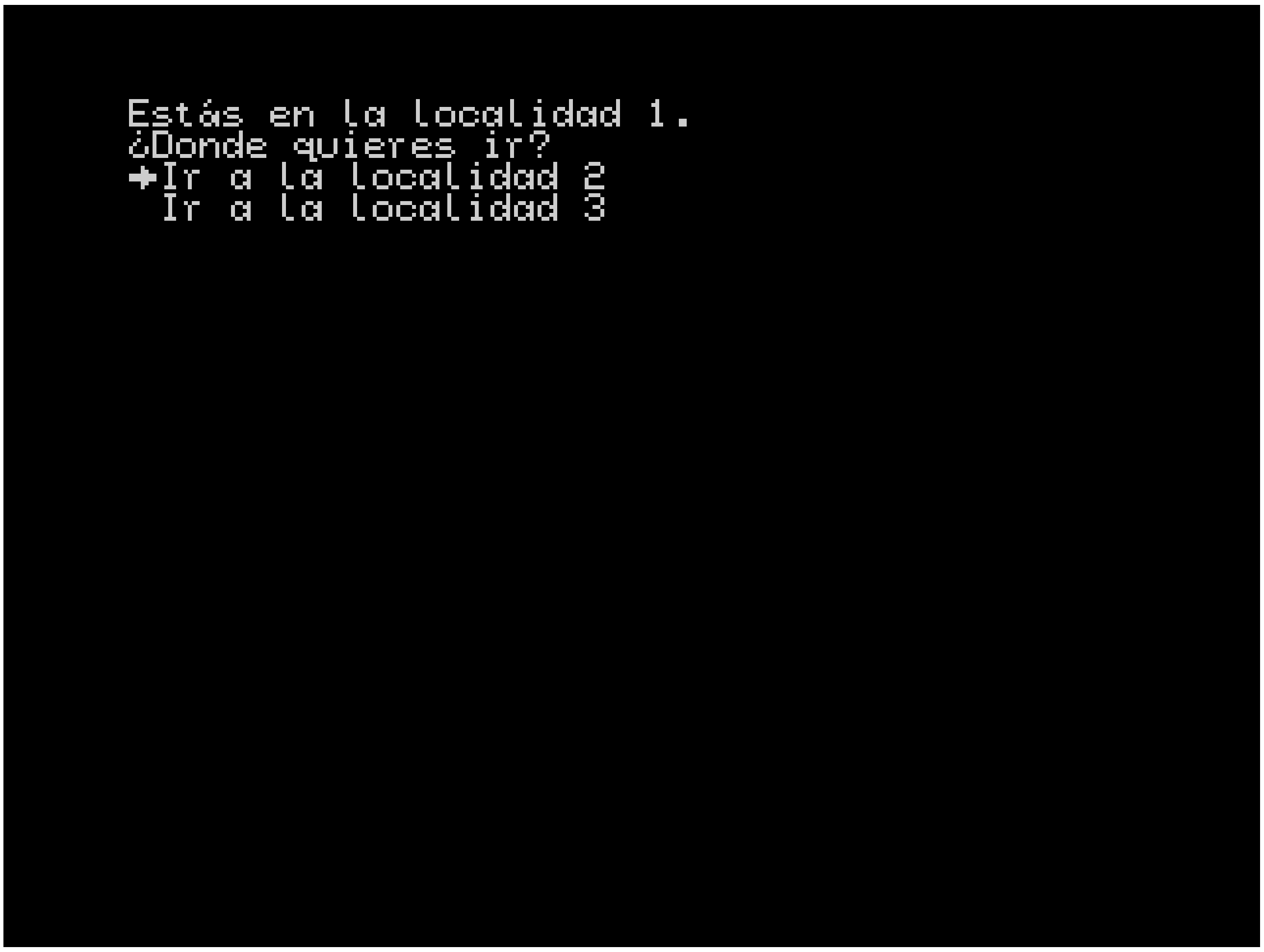
Options

Options are the most important feature of the engine. Again, let's see it with the example from the manual:

```
[[ /* Pone colores de pantalla y la borra */
   PAPER 0    /* Color de fondo negro  */
   INK   7    /* Color de texto blanco */
   BORDER 0   /* Borde de color negro  */
```

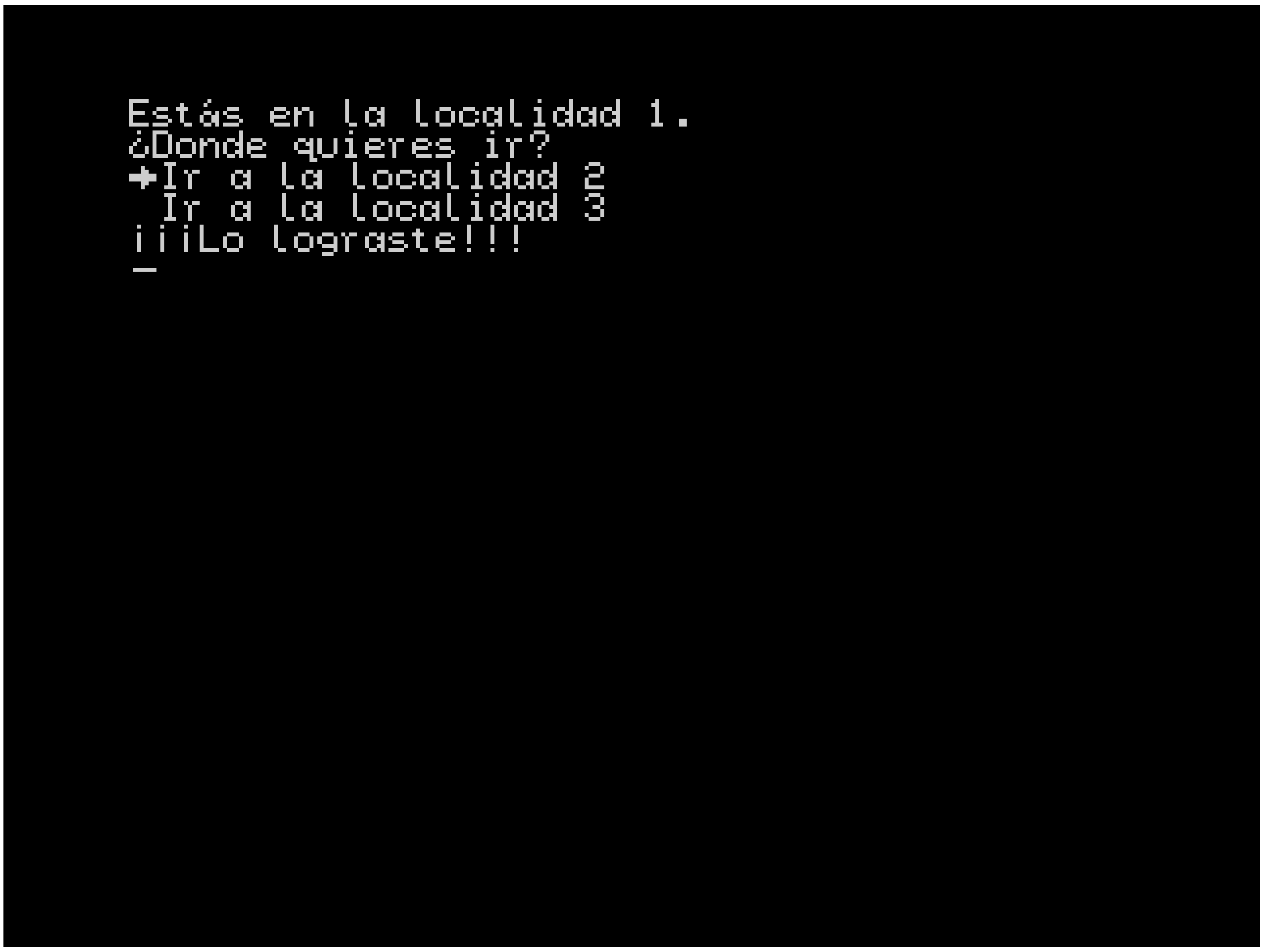
```
    CLEAR      /* Borramos la pantalla*/
]][[ LABEL Localidad1]]Estás en la localidad 1. ¿Donde quieres ir?
[[ OPTION GOTO Localidad2 ]]Ir a la localidad 2
[[ OPTION GOTO Localidad3 ]]Ir a la localidad 3
[[ CHOOSE ]]
[[ LABEL Localidad2 ]]¡¡¡Lo lograste!!!
[[ GOTO Final ]]
[[ LABEL Localidad3 ]]¡¡¡Estas muerto!!!
[[ GOTO Final]]
[[ LABEL Final : WAITKEY: END ]]
```

When compiling and running we have this:



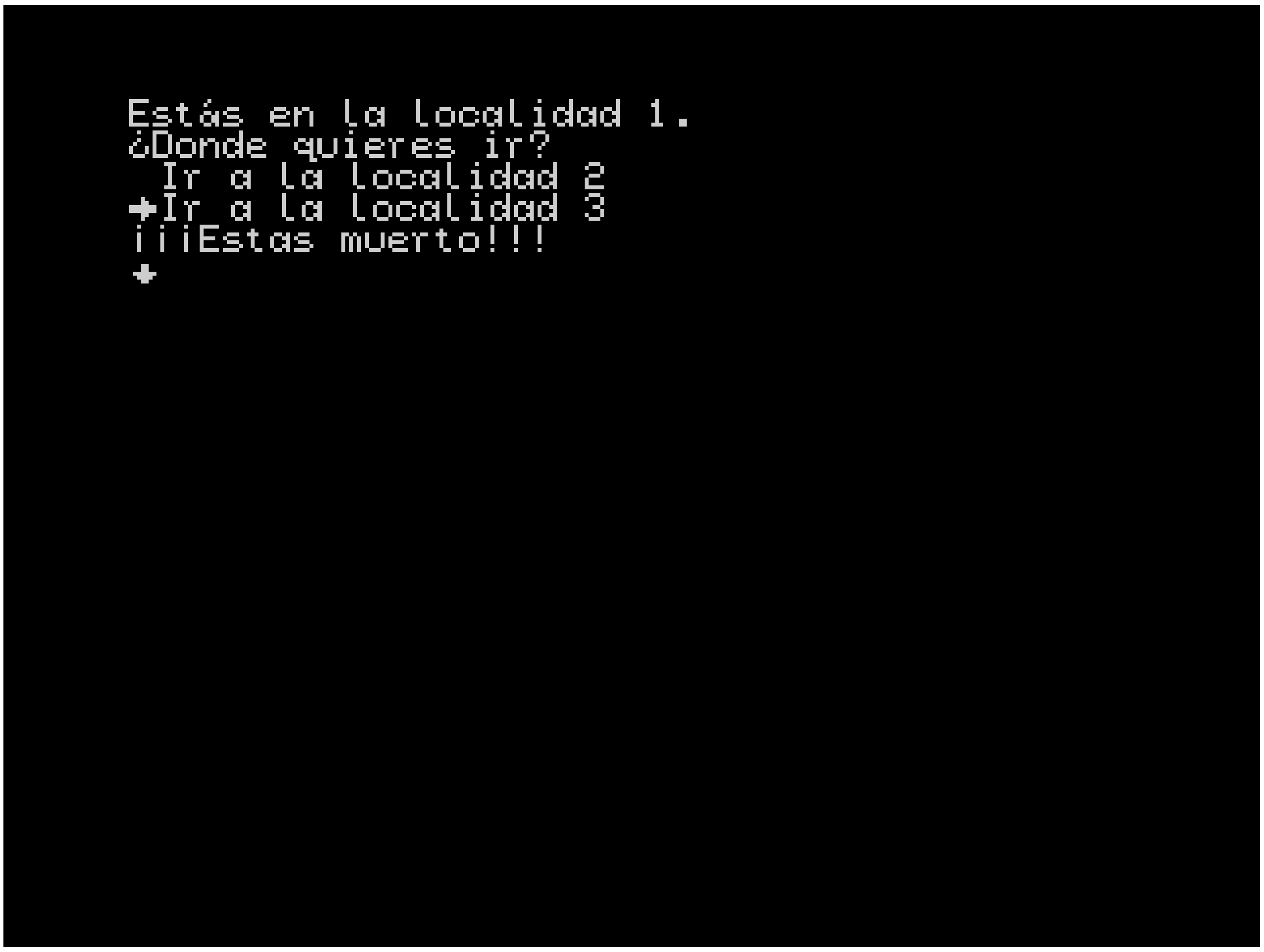
```
Estás en la localidad 1.
¿Donde quieres ir?
➔Ir a la localidad 2
 Ir a la localidad 3
```

We are shown two options that we can choose with the **P** and **Q** keys and select one with **Space** or **Enter**. If we choose the first option, we get this:



```
Estás en la localidad 1.
¿Donde quieres ir?
➜Ir a la localidad 2
 Ir a la localidad 3
¡¡¡Lo lograste!!!
_
```

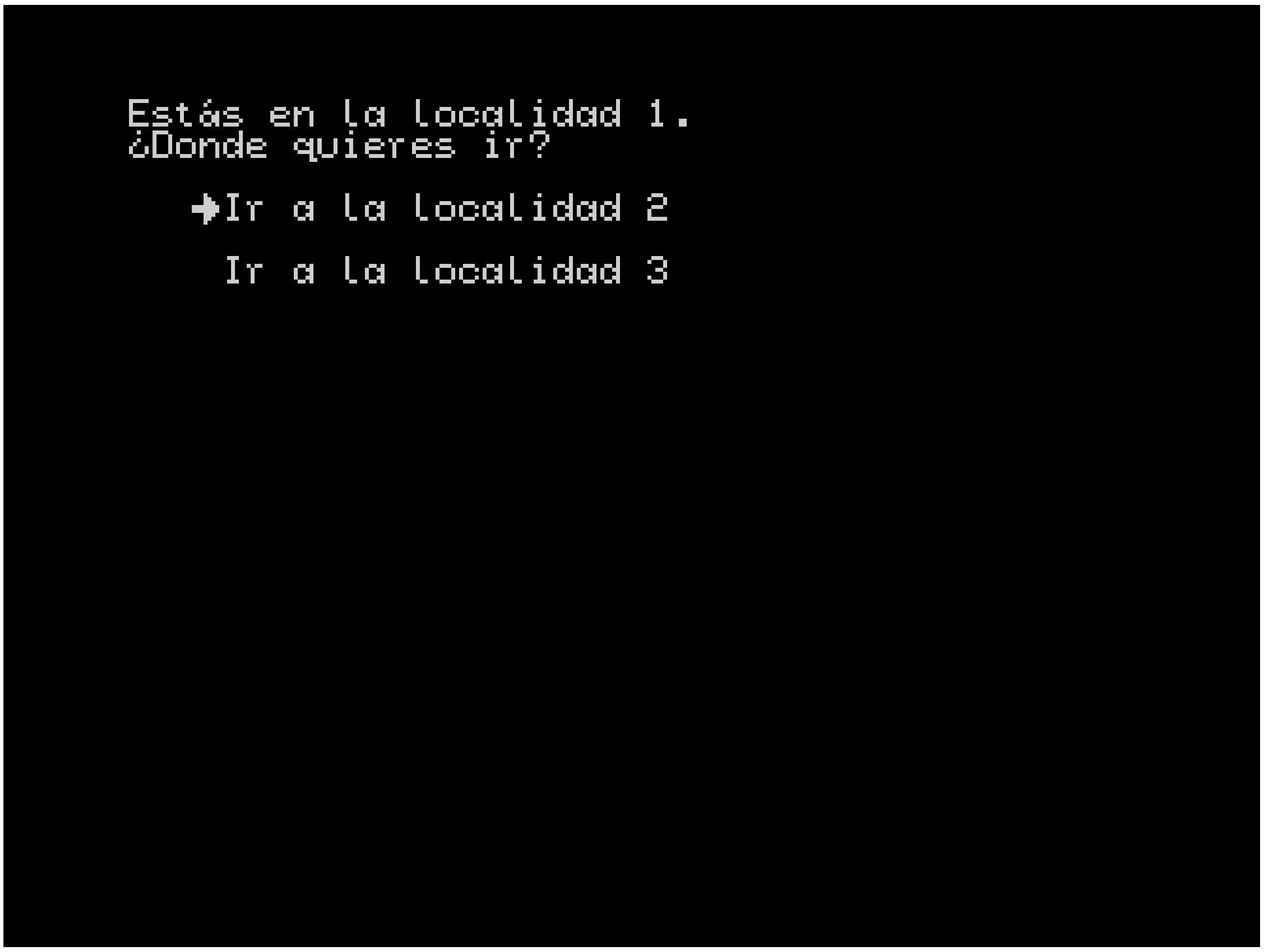
And if we choose the second:



```
Estás en la localidad 1.
¿Donde quieres ir?
 Ir a la localidad 2
➜Ir a la localidad 3
¡¡¡Estas muerto!!!
```

With the command `OPTION GOTO label`, what we do is declare a selectable option. The place where the cursor is at that moment will be the point where the option icon appears. Let's rearrange the options a bit to illustrate this last point:

```
[[ /* Pone colores de pantalla y la borra */
   PAPER 0    /* Color de fondo negro  */
   INK   7    /* Color de texto blanco */
   BORDER 0   /* Borde de color negro  */
   CLEAR      /* Borramos la pantalla*/
]][[ LABEL Localidad1]]Estás en la localidad 1.
¿Donde quieres ir?

  [[ OPTION GOTO Localidad2 ]]Ir a la localidad 2

  [[ OPTION GOTO Localidad3 ]]Ir a la localidad 3
[[ CHOOSE ]]
[[ LABEL Localidad2 ]]¡¡¡Lo lograste!!!
[[ GOTO Final ]]
[[ LABEL Localidad3 ]]¡¡¡Estas muerto!!!
[[ GOTO Final]]
[[ LABEL Final : WAITKEY: END ]]
```



```
Estás en la localidad 1.
¿Donde quieres ir?

  ➜Ir a la localidad 2

    Ir a la localidad 3
```

As you can see, we have separated the options with line breaks and put two spaces of indentation before the `OPTION GOTO` command and this is reflected in the final result.

Once the options have been declared, with the `CHOOSE` command, we activate the menu, which will allow us to choose between one of the options that were already on the screen. When we select one, it will jump to the label indicated in the corresponding `OPTION GOTO`. In the example, if we select the first option, `OPTION GOTO Localidad2`, it will jump to the label `LABEL Localidad2` and print *You did it* and then jump to the label `LABEL Final`. The `GOTO Final` is necessary, because if not, it would print *You did it!!!* and then *You are dead!!!*; the `GOTO` is necessary, in this case, to avoid the result of the second option being executed.

As an additional note with `CHOOSE`, only a maximum of 32 options and a minimum of one are allowed (useless, but allowed). Outside that range the interpreter will give an error.

Also note that there is a timed variant of `CHOOSE`, compile and run this without selecting anything from the menu:

```
[[ /* Pone colores de pantalla y la borra */
   PAPER 0    /* Color de fondo negro  */
   INK   7    /* Color de texto blanco */
   BORDER 0   /* Borde de color negro  */
   CLEAR      /* Borramos la pantalla*/
]][[ LABEL Localidad1]]Estás en la localidad 1.
¿Donde quieres ir?

  [[ OPTION GOTO Localidad2 ]]Ir a la localidad 2

  [[ OPTION GOTO Localidad3 ]]Ir a la localidad 3
[[ CHOOSE IF WAIT 500 THEN GOTO Localidad3]]
[[ LABEL Localidad2 ]]¡¡¡Lo lograste!!!
[[ GOTO Final ]]
[[ LABEL Localidad3 ]]¡¡¡Estas muerto!!!
[[ GOTO Final]]
[[ LABEL Final : WAITKEY: END ]]
```

You will notice that after about 10 seconds, it has displayed *You are dead!!!*.

What `CHOOSE IF WAIT 500 THEN GOTO Location3` does is the same as `CHOOSE`, activate the selection of options, but with the exception that it also performs a countdown, in this case from 500. If this countdown reaches zero without anything being selected, then the jump is made to the indicated label; in this case *Location3*. The counter works based on the Spectrum frames, that is, 1/50 of a second, so 500/50 = 10 seconds. We will see this in the next chapter.

Again, we could rewrite the previous code like this with the shortened form of the labels:

```
[[ /* Pone colores de pantalla y la borra */
   PAPER 0    /* Color de fondo negro  */
   INK   7    /* Color de texto blanco */
   BORDER 0   /* Borde de color negro  */
   CLEAR      /* Borramos la pantalla*/
]][[ #Localidad1 ]]Estás en la localidad 1.
¿Donde quieres ir?
```

```
    [[ OPTION GOTO Localidad2 ]]Ir a la localidad 2

    [[ OPTION GOTO Localidad3 ]]Ir a la localidad 3
  [[ CHOOSE IF WAIT 500 THEN GOTO Localidad3]]
  [[ #Localidad2 ]]¡¡¡Lo lograste!!!
  [[ GOTO Final ]]
  [[ #Localidad3 ]]¡¡¡Estas muerto!!!
  [[ GOTO Final]]
  [[ #Final : WAITKEY: END ]]
```

From now on, I'll be using the shortened form for labels.

With this we already have the basis for making a basic "Choose Your Own Adventure". But we still have many more possibilities to explore...

Pauses and Waits

In the previous examples you will have already seen the `WAITKEY` command. This command generates a pause, with an animated icon, waiting for the confirmation key to be pressed. With this you can control the display of the text and avoid the user having to read a wall of text in one sitting, as well as allowing you to control the presentation.

An example would be when the "page" is ending and we want the user to press a key to move to the next one, which we can do like this:

```
Texto al final de la página.[[
  WAITKEY
  CLEAR
]]Texto al principio de la siguiente página.
```

With the `WAITKEY` we wait, and when we confirm, with the following `CLEAR` we erase the text on the screen and start writing from the beginning of what would be the next "page".

However, there is an option for CYD to do this on its own. The default behavior when we finish printing on the last line is to completely clear the screen and continue writing; but with the `PAGEPAUSE` command we can activate an alternative behavior. If we indicate `PAGEPAUSE 1`, for example, when the available space runs out, it will automatically generate a wait for the user to press the confirmation key before clearing the screen and continuing printing.

In addition, there are also "timed" waits, being `CHOOSE IF WAIT X THEN GOTO Y` from the previous chapter an example. When they are executed, a counter is loaded with the value passed as a parameter and a countdown is performed until the counter reaches zero. The counter is decremented once every Spectrum frame, that is, once every 1/50 of a second, or in other words, 50 times per second. So, if we want to wait a second, we have to set the counter to 50.

With this, we already have what is necessary to know the commands:

- With the `WAIT` command, an unconditional wait is performed, it is a stop until the counter runs out.

```
Espera tres segundos[[WAIT 150]]
Ya está[[WAITKEY]]
```

- The `PAUSE` command is a combination of `WAITKEY` and `WAIT`, a wait is performed until the counter runs out or the user presses the confirmation key, we could consider it a `WAITKEY` with expiration.

```
Espera tres segundos o pulsa una tecla[[PAUSE 150]]
Ya está[[WAITKEY]]
```

- `CHOOSE IF WAIT X THEN GOTO Y` has already been explained in the previous chapter. If the counter runs out before a menu option is selected, the indicated jump is made.

Finally, let's talk about the `TYPERATE` command, which is a bit special compared to the rest of the wait commands. With this command we indicate the wait that occurs each time a character is printed. This wait is not adjusted to frames, but is a counter that depends on the speed of the processor (faster). The idea of this command is to write in a slower and more gradual way, for certain "dramatic" situations.

```
[[TYPERATE 100]]Esto imprime lento
[[TYPERATE 0]]Esto imprime normal[[WAITKEY]]
```

Text layout on screen

One of the most important parts of designing a CYD adventure is adjusting the presentation of the text. CYD doesn't "know" how to present the text. As authors, we have to help it.

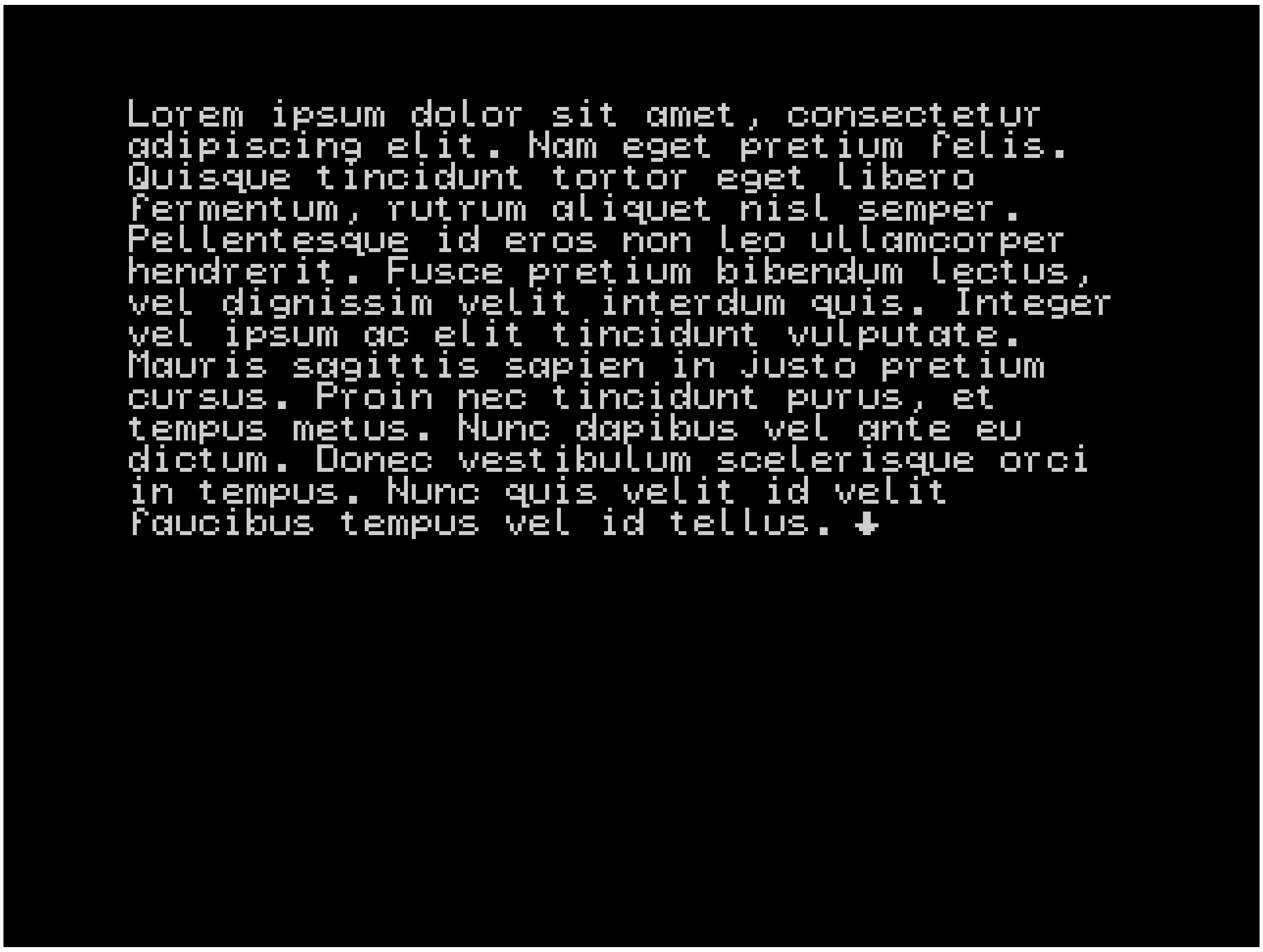
We can visualize its behavior by imagining that there is an invisible cursor on the screen that prints the text from left to right and from top to bottom. The engine always tries to ensure that the words do not break, so that if the next word does not fit on the remaining line, it jumps to the next line and prints it there. Any text separated by spaces is considered a word.

Let's put a text like *Loren ipsum*, all in one go, without line breaks:

```
[[ /* Pone colores de pantalla y la borra */
   PAPER 0    /* Color de fondo negro  */
   INK   7    /* Color de texto blanco */
   BORDER 0   /* Borde de color negro  */
   CLEAR      /* Borramos la pantalla*/
   PAGEPAUSE 1
]]Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nam eget pretium felis.
Quisque tincidunt tortor eget libero fermentum, rutrum aliquet nisl semper.
Pellentesque id eros non leo ullamcorper hendrerit. Fusce pretium bibendum lectus,
vel dignissim velit interdum quis. Integer vel ipsum ac elit tincidunt vulputate.
Mauris sagittis sapien in justo pretium cursus. Proin nec tincidunt purus, et
tempus metus. Nunc dapibus vel ante eu dictum. Donec vestibulum scelerisque orci
```

```
in tempus. Nunc quis velit id velit faucibus tempus vel id tellus.[[ WAITKEY: END
]]
```

This is the result:



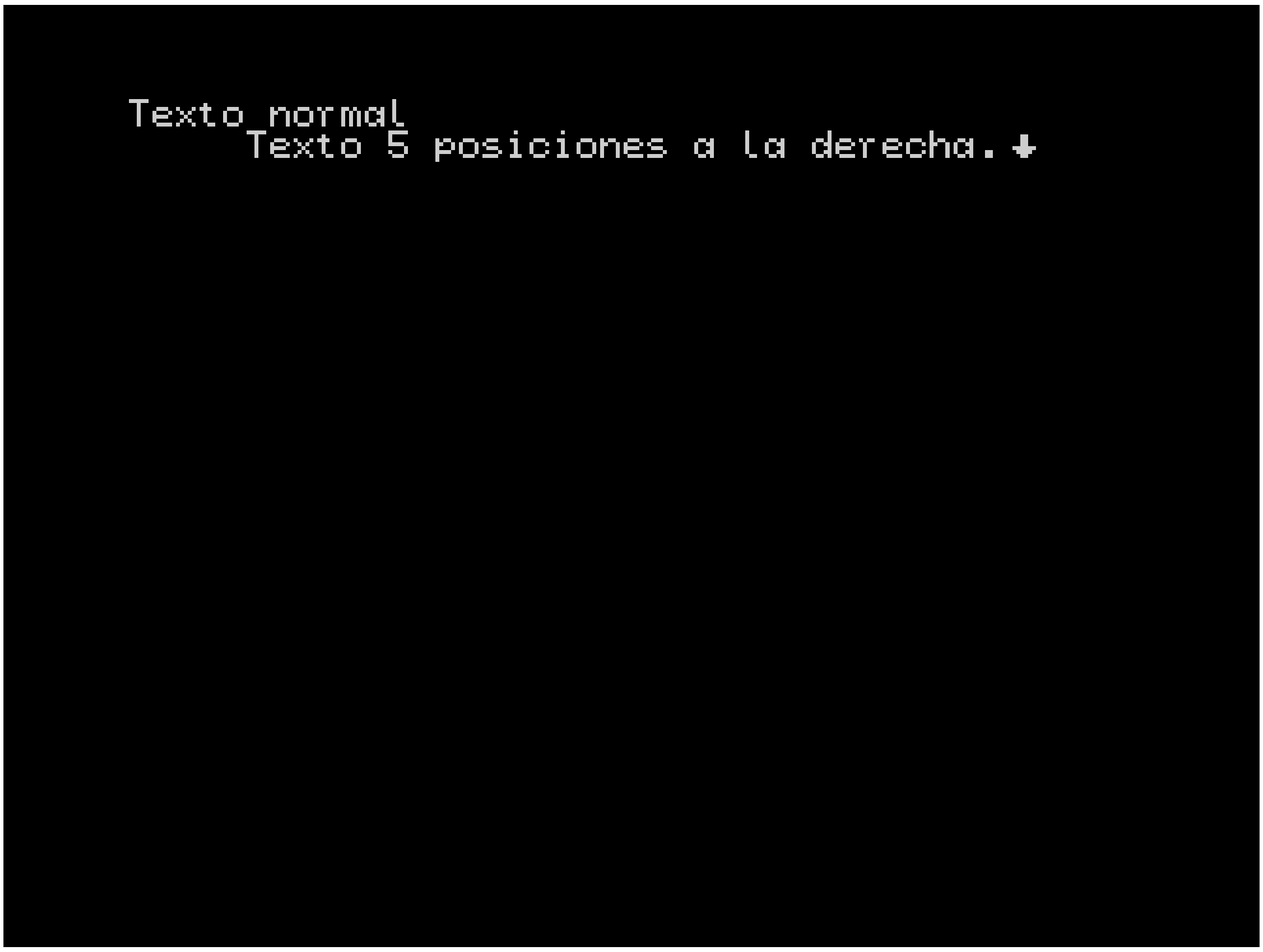
```
Lorem ipsum dolor sit amet, consectetur
adipiscing elit. Nam eget pretium felis.
Quisque tincidunt tortor eget libero
fermentum, rutrum aliquet nisl semper.
Pellentesque id eros non leo ullamcorper
hendrerit. Fusce pretium bibendum lectus,
vel dignissim velit interdum quis. Integer
vel ipsum ac elit tincidunt vulputate.
Mauris sagittis sapien in justo pretium
cursus. Proin nec tincidunt purus, et
tempus metus. Nunc dapibus vel ante eu
dictum. Donec vestibulum scelerisque orci
in tempus. Nunc quis velit id velit
faucibus tempus vel id tellus. ⬇
```

You can see that words that do not fit on their line continue on the next line without being cut off.

When the print cursor reaches the last line and must move to the next line, the screen is cleared and printing continues from the origin of the screen, at the top left; except if you use the `PAGEPAUSE` command, which generates a confirmation wait before clearing the screen.

We can layout the screen properly using spaces and line breaks as needed, but if we want to make an indent or tabs, you have the `TAB pos` command, which prints as many spaces as indicated in the parameter:

```
[[ /* Pone colores de pantalla y la borra */
   PAPER 0    /* Color de fondo negro  */
   INK   7    /* Color de texto blanco */
   BORDER 0   /* Borde de color negro  */
   CLEAR      /* Borramos la pantalla*/
   PAGEPAUSE 1
]]Texto normal
[[TAB 5]]Texto 5 posiciones a la derecha.[[ WAITKEY: END ]]
```

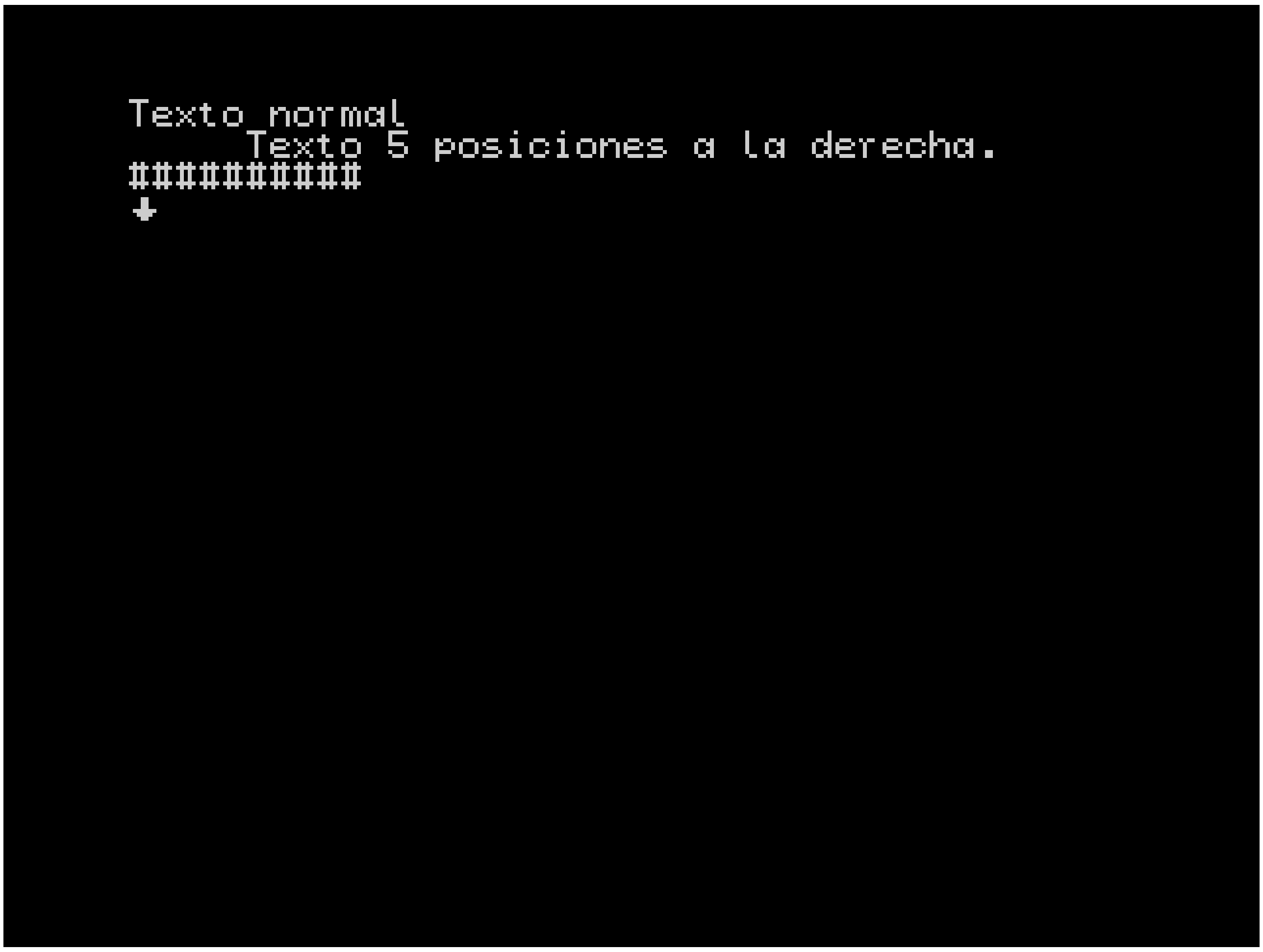


```
Texto normal
     Texto 5 posiciones a la derecha.
```

The TAB command allows us to save memory because, if we want to print 10 spaces, those 10 spaces will consume more than using the `TAB 10` command.

Similarly, we have the REPCHAR command to print any character repeatedly:

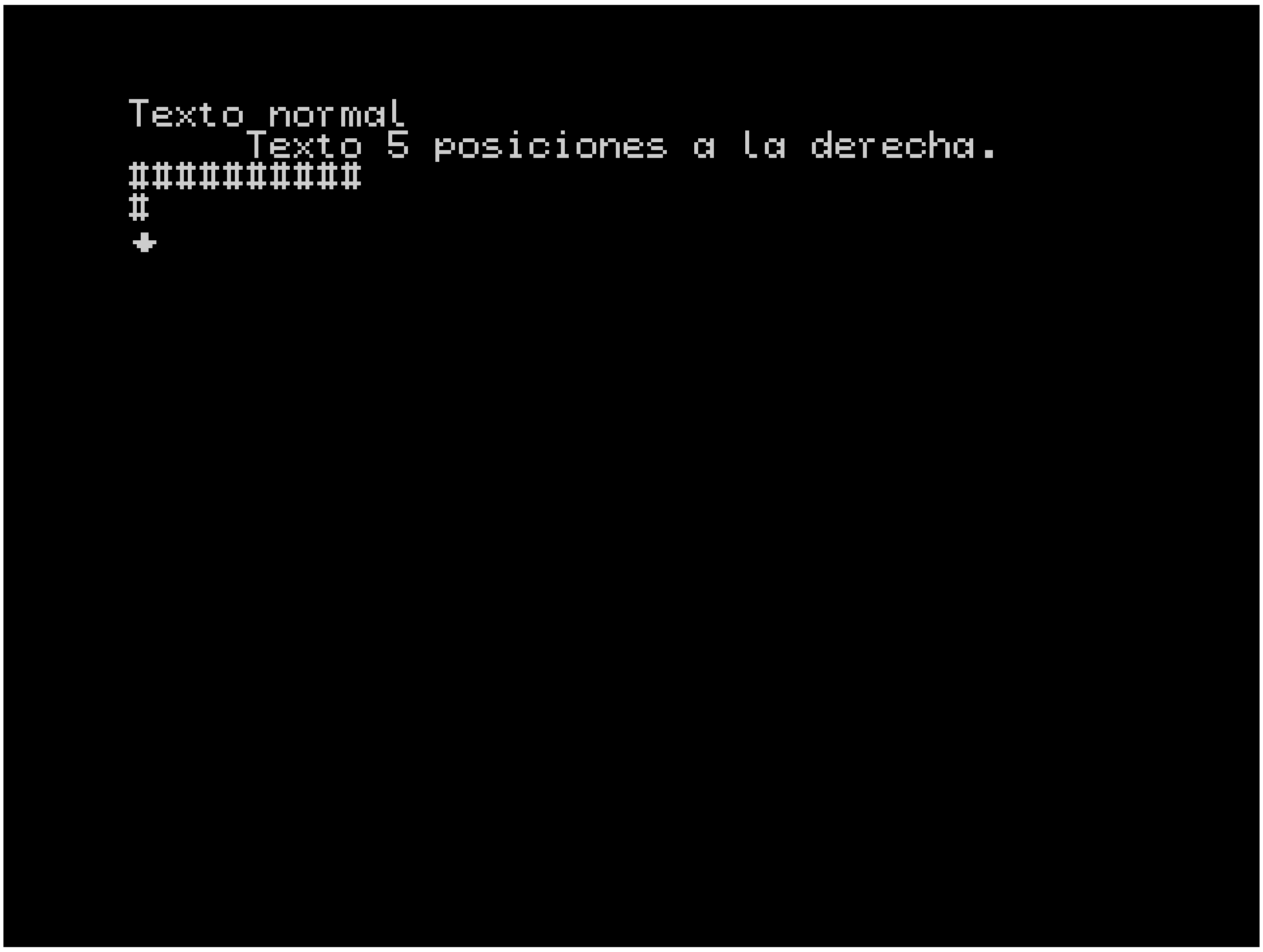
```
[[ /* Pone colores de pantalla y la borra */
   PAPER 0    /* Color de fondo negro  */
   INK   7    /* Color de texto blanco */
   BORDER 0   /* Borde de color negro  */
   CLEAR      /* Borramos la pantalla*/
   PAGEPAUSE 1
]]Texto normal
[[ TAB 5 ]]Texto 5 posiciones a la derecha.
[[ REPCHAR 35, 10 ]]
[[ WAITKEY: END ]]
```



```
Texto normal
     Texto 5 posiciones a la derecha.
##########
```

This repeats the number 35 (the pad character), 10 times. And if we want to print it only once, we have the CHAR command:

```
[[ /* Pone colores de pantalla y la borra */
   PAPER 0    /* Color de fondo negro  */
   INK   7    /* Color de texto blanco */
   BORDER 0   /* Borde de color negro  */
   CLEAR      /* Borramos la pantalla*/
   PAGEPAUSE 1
]]Texto normal
[[ TAB 5 ]]Texto 5 posiciones a la derecha.
[[ REPCHAR 35, 10 ]]
[[ CHAR 35 ]]
[[ WAITKEY: END ]]
```



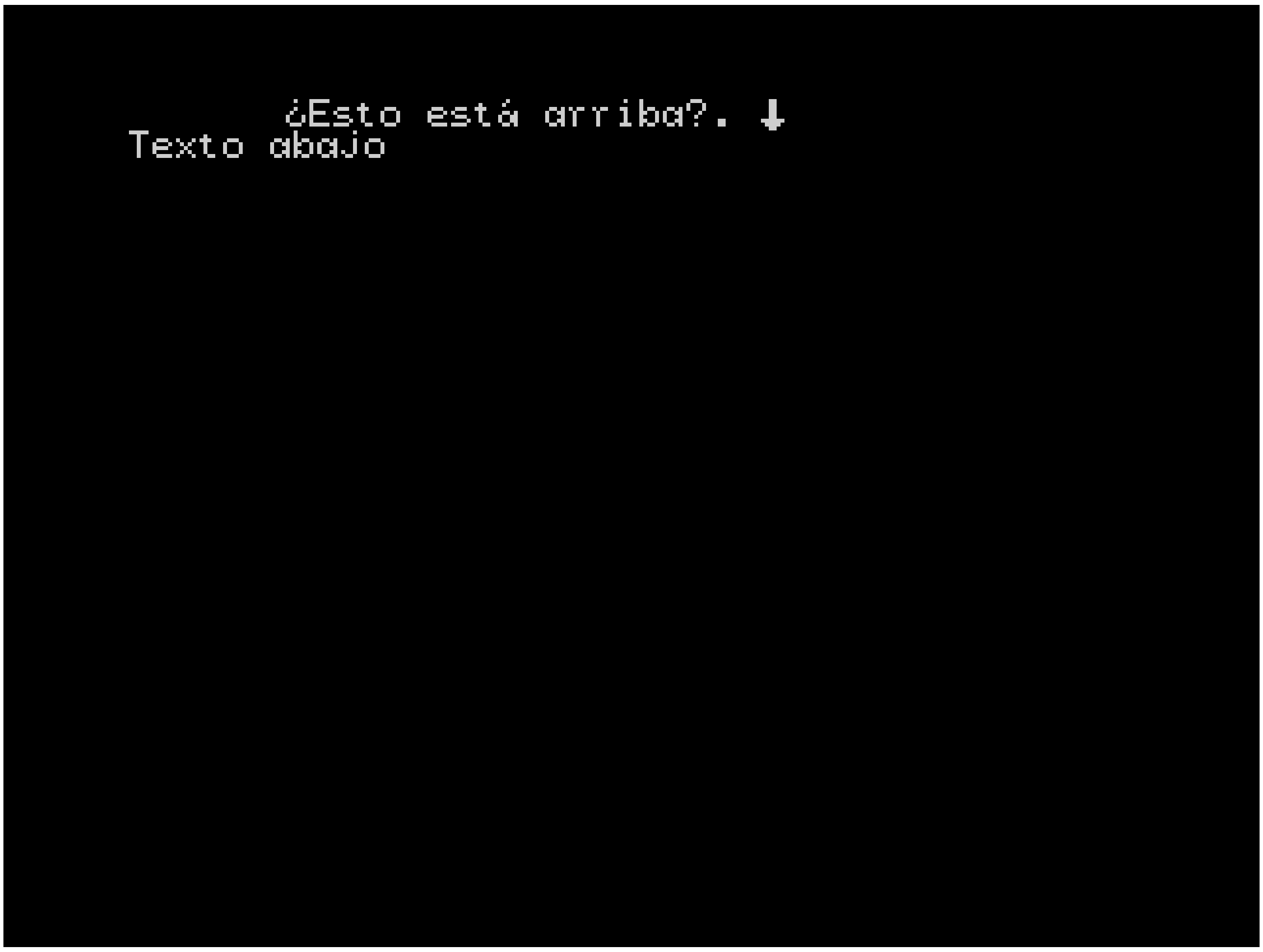
```
Texto normal
     Texto 5 posiciones a la derecha.
##########
#
```

And to make a line break, the `NEWLINE` command:

```
[[ /* Pone colores de pantalla y la borra */
   PAPER 0    /* Color de fondo negro  */
   INK   7    /* Color de texto blanco */
   BORDER 0   /* Borde de color negro  */
   CLEAR      /* Borramos la pantalla*/
   PAGEPAUSE 1
]]Texto normal
[[ TAB 5 ]]Texto 5 posiciones a la derecha.
[[ NEWLINE : REPCHAR 35, 10 ]]
[[ CHAR 35 ]]
[[ WAITKEY: END ]]
```

Let's now look at a command to place the print cursor anywhere on the screen. With the command `AT column,row`, we can indicate the coordinates where we want to place the cursor to continue printing. Let's see it with this example:

```
[[ /* Pone colores de pantalla y la borra */
   PAPER 0    /* Color de fondo negro  */
   INK   7    /* Color de texto blanco */
   BORDER 0   /* Borde de color negro  */
   CLEAR      /* Borramos la pantalla*/
   PAGEPAUSE 1
]]
Texto abajo
[[AT 5,0]]¿Esto está arriba?.[[ WAITKEY: END ]]
```


¿Esto está arriba?. ⬇
Texto abajo

What happened here? If we examine the code, we see that before "Text below", there is a line break. So the cursor jumps to the next line and prints the "Text below", but after that we have `AT 5,0`, which means *move the cursor to column 5 and row 0*, that is, it goes back to the previous row and 5 positions to the right from the origin.

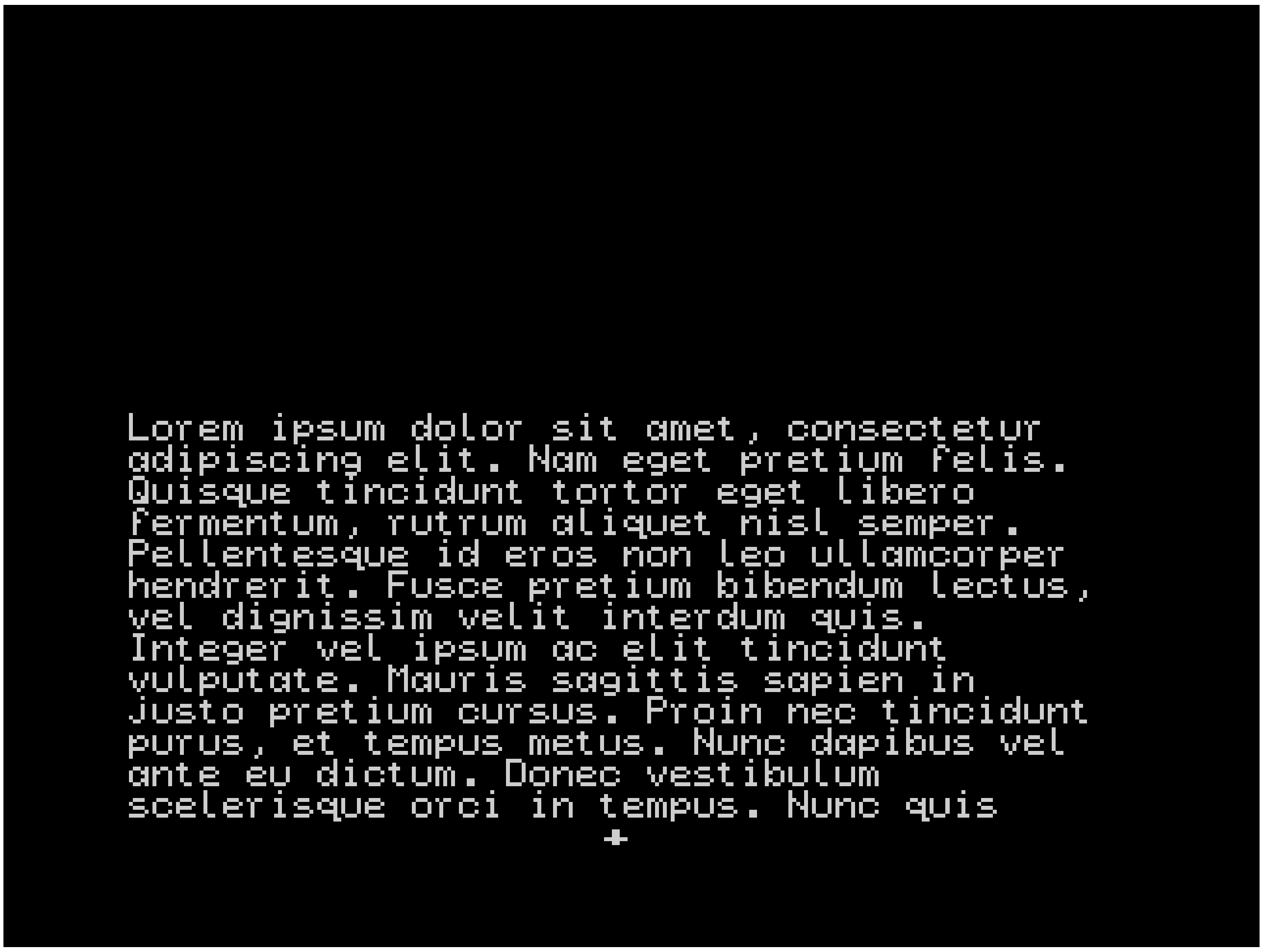
With this we can now place text wherever we want. But we are missing something to fully control the layout of the text on the screen, and that is to define some margins. By default CYD prints the text in full screen, but we may want it to only print in a certain area so as not to "cover" images that we want to show. For this we have the command `MARGINS`, which allows us to indicate the "rectangle" or printing area of the texts.

The format of the command is `MARGINS origin_col, origin_row, width, height`, where the parameters are the origin column and row of the print area and the corresponding width and height. By default, the engine starts as if the command `MARGINS 0, 0, 32, 24` had been executed, that is, the origin at the top left side of the screen and the size at the full screen.

Now let's go back to the initial example of this chapter and put it in the lower area of the screen:

```
[[ /* Pone colores de pantalla y la borra */
   PAPER 0    /* Color de fondo negro  */
   INK   7    /* Color de texto blanco */
   BORDER 0   /* Borde de color negro  */
   CLEAR      /* Borramos la pantalla*/
   PAGEPAUSE 1
   MARGINS 0, 10, 32, 14
]]Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nam eget pretium felis.
Quisque tincidunt tortor eget libero fermentum, rutrum aliquet nisl semper.
Pellentesque id eros non leo ullamcorper hendrerit. Fusce pretium bibendum lectus,
vel dignissim velit interdum quis. Integer vel ipsum ac elit tincidunt vulputate.
Mauris sagittis sapien in justo pretium cursus. Proin nec tincidunt purus, et
tempus metus. Nunc dapibus vel ante eu dictum. Donec vestibulum scelerisque orci
in tempus. Nunc quis velit id velit faucibus tempus vel id tellus.[[ WAITKEY: END
]]
```

We have lowered the origin of the print area to row 10, and reduced its height to 14:



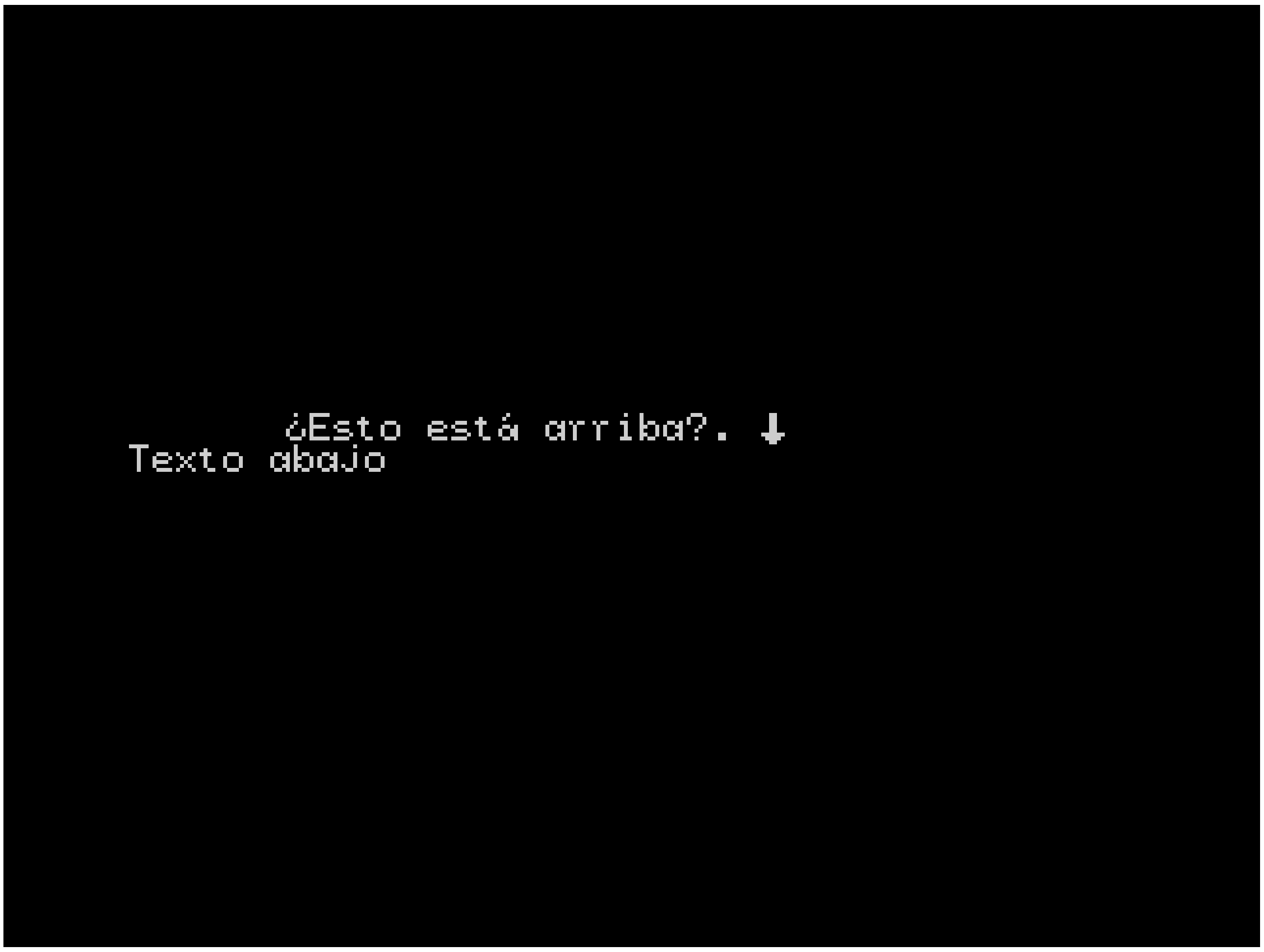
```
Lorem ipsum dolor sit amet, consectetur
adipiscing elit. Nam eget pretium felis.
Quisque tincidunt tortor eget libero
fermentum, rutrum aliquet nisl semper.
Pellentesque id eros non leo ullamcorper
hendrerit. Fusce pretium bibendum lectus,
vel dignissim velit interdum quis.
Integer vel ipsum ac elit tincidunt
vulputate. Mauris sagittis sapien in
justo pretium cursus. Proin nec tincidunt
purus, et tempus metus. Nunc dapibus vel
ante eu dictum. Donec vestibulum
scelerisque orci in tempus. Nunc quis
```

With this we can adjust the area where we want to print. Note the effect of `PAGEPAUSE 1`, which, since not everything fits, generates a confirmation icon in the center.

Now we are going to use the same thing in the second example of this chapter:

```
[[ /* Pone colores de pantalla y la borra */
   PAPER 0    /* Color de fondo negro  */
   INK   7    /* Color de texto blanco */
   BORDER 0   /* Borde de color negro  */
   CLEAR      /* Borramos la pantalla*/
   PAGEPAUSE 1
   MARGINS 0, 10, 32, 14
]]
Texto abajo
[[AT 5,0]]¿Esto está arriba?.[[ WAITKEY: END ]]
```

Do you notice anything strange?



```
          ¿Esto está arriba?. ⬇
Texto abajo
```

If you haven't noticed, the coordinates of `AT` are not, in this case, the screen coordinates. The coordinates of `AT` are **always relative to the origin of the print area**. When we had the area defined as the full screen, it corresponded to the screen coordinates, (0,0). Now they are relative to the new origin, (0,10), which would send the cursor to position (5,10) on the screen.

The Spectrum screen has, naturally, 32 characters per line, which is somewhat insufficient for long texts. CYD supports variable width fonts and the default one uses 6x8 characters, which allows us to have 42 characters per line, but this causes it to not match the Spectrum attribute grid, which is 8x8, and there is "colour clash".

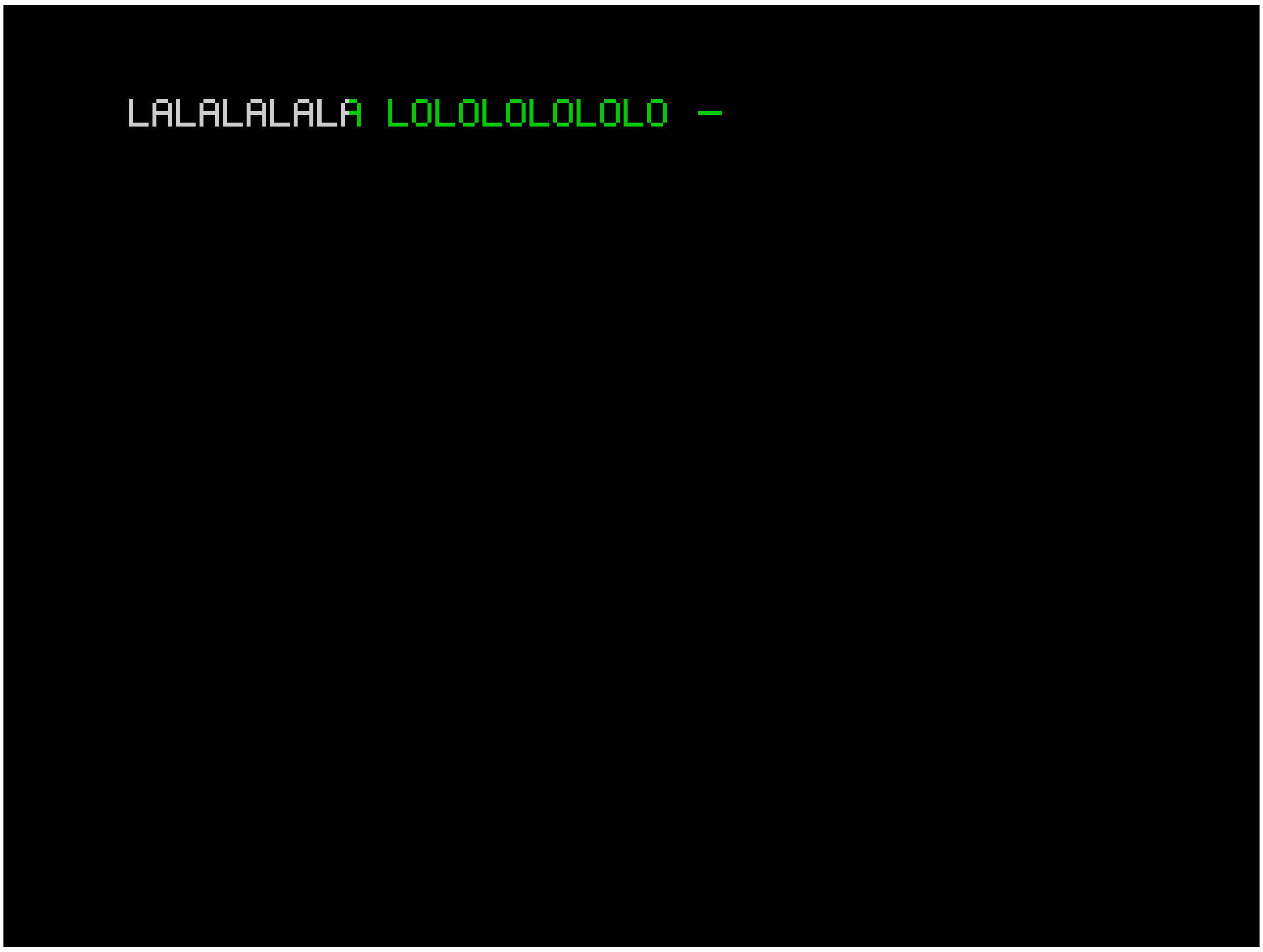
For this reason, `AT` and `MARGINS` have a character-based coordinate system of 8x8 pixels, that is, 32 columns and 24 rows, counted from 0 to 31 and 0 to 23 respectively.

Finally, let's look at an attribute collision problem with this example:

```
[[ /* Pone colores de pantalla y la borra */
   PAPER 0    /* Color de fondo negro  */
   BORDER 0   /* Borde de color negro  */
```

```
    CLEAR      /* Borramos la pantalla*/
    PAGEPAUSE 1
    INK    7    /* Color de texto azul */
]]LALALALALA[[INK 4 /* COlor verde */]] LOLOLOLOLOLO[[ WAITKEY: END ]]
```

You can see that when you change the color to green with the `INK` command, because the next character to be printed (a space) is between two attribute cells, the previous character is painted white:

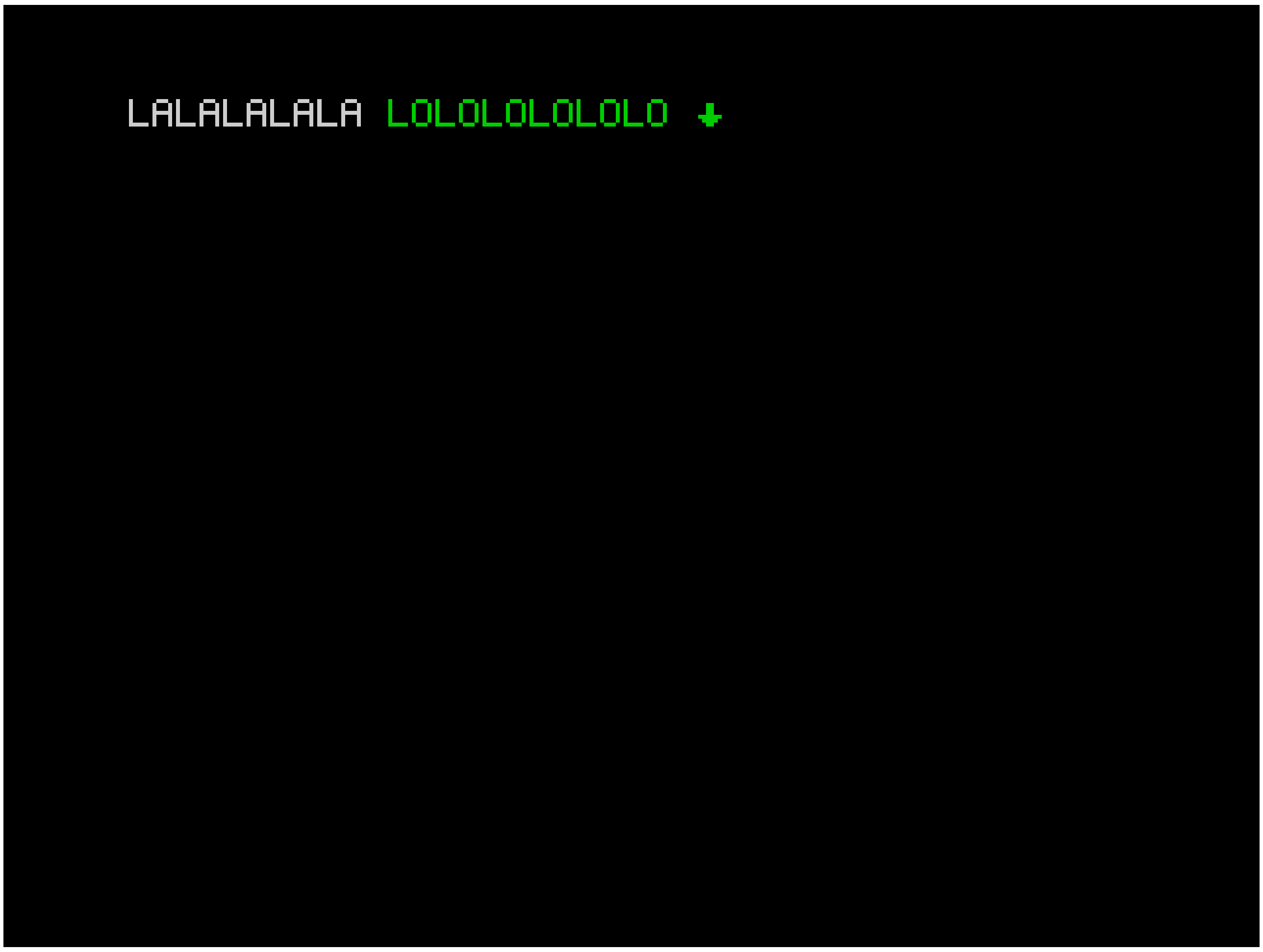


```
LALALALALA LOLOLOLOLOLO _
```

We can solve this by changing the color after the space:

```
[[ /* Pone colores de pantalla y la borra */
    PAPER 0     /* Color de fondo negro  */
    BORDER 0    /* Borde de color negro  */
    CLEAR       /* Borramos la pantalla*/
    PAGEPAUSE 1
    INK    7    /* Color de texto blanco */
]]LALALALALA [[INK 4 /*COlor verde */]]LOLOLOLOLOLO[[ WAITKEY: END ]]
```

And now it's correct:



LALALALALA LOLOLOLOLOLO ⬇

The author will need to make these small adjustments to improve the presentation of the text. If you want to avoid them completely, do not mix different colors within the same line.

Images

To make the adventure more attractive, it is possible to add images in SCR format, of Spectrum screens. These images will be compressed with the `CSC` utility, creating files with the same extension, which must be included in the final disk.

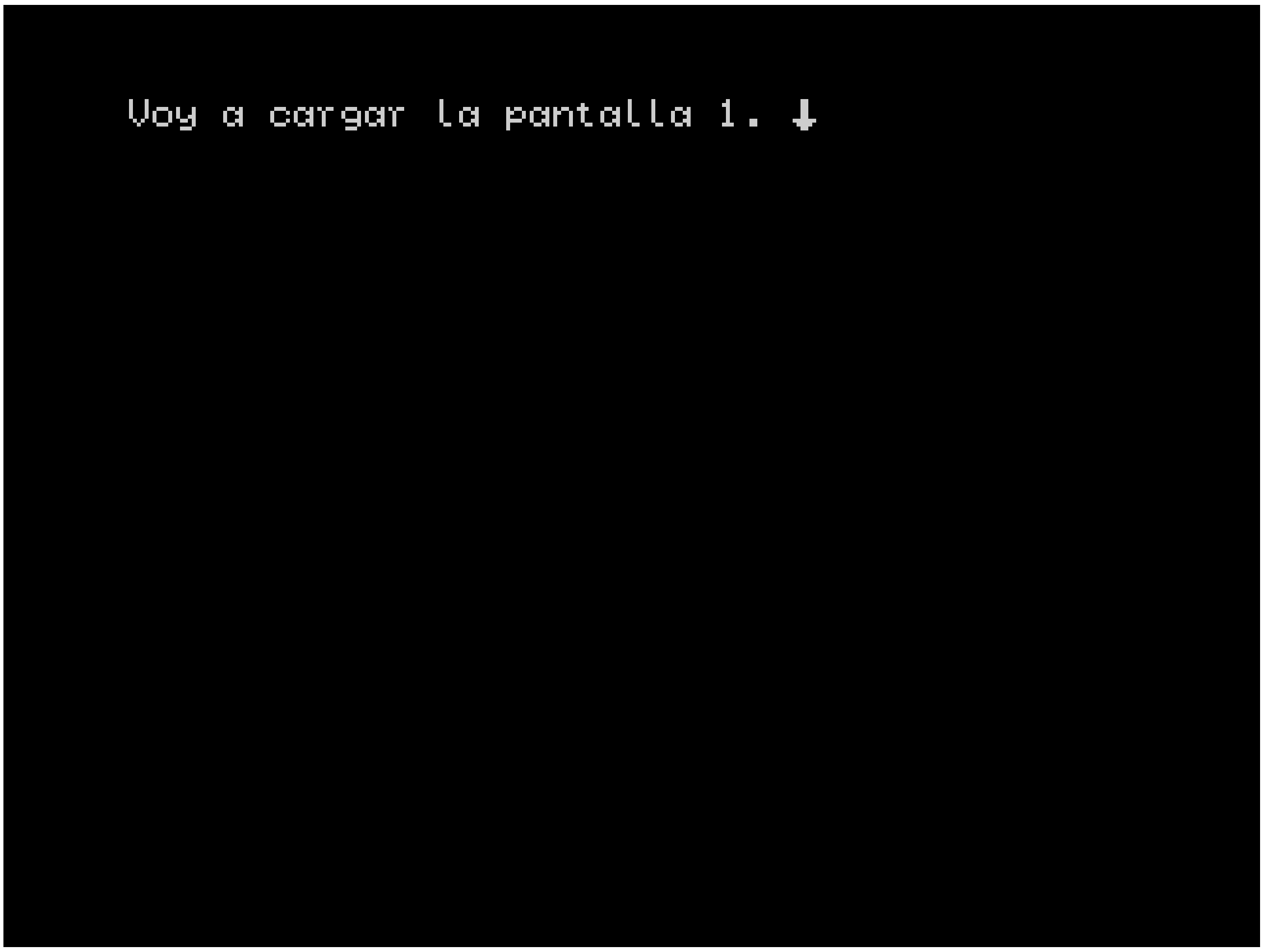
The manual explains how `CSC` works, in case you want to do it manually. But the `MakeAdv` script automatically searches for and compresses the SCR files that are in the `\IMAGES` directory, so you will simply have to place the files there.

We are going to use an image that we have as an example, called `ORIGIN1.SCR`, inside the `\examples\test\IMAGES` directory. Copy it and rename it as `001.SCR`. And put the following code in `tutorial.txt`:

```
[[ /* Pone colores de pantalla y la borra */
   PAPER 0    /* Color de fondo negro  */
   BORDER 0   /* Borde de color negro  */
   CLEAR      /* Borramos la pantalla*/
   INK   7    /* Color de texto blanco */
   PAGEPAUSE 1
]]Voy a cargar la pantalla 1.[[
   WAITKEY
   /* Cargamos la imagen del fichero 001.CSC */
   PICTURE 1]]
Voy a mostrar la imagen.[[
   WAITKEY
   /* Cargamos la imagen cargada  */
   DISPLAY 1
]]
Hecho[[ WAITKEY: END ]]
```

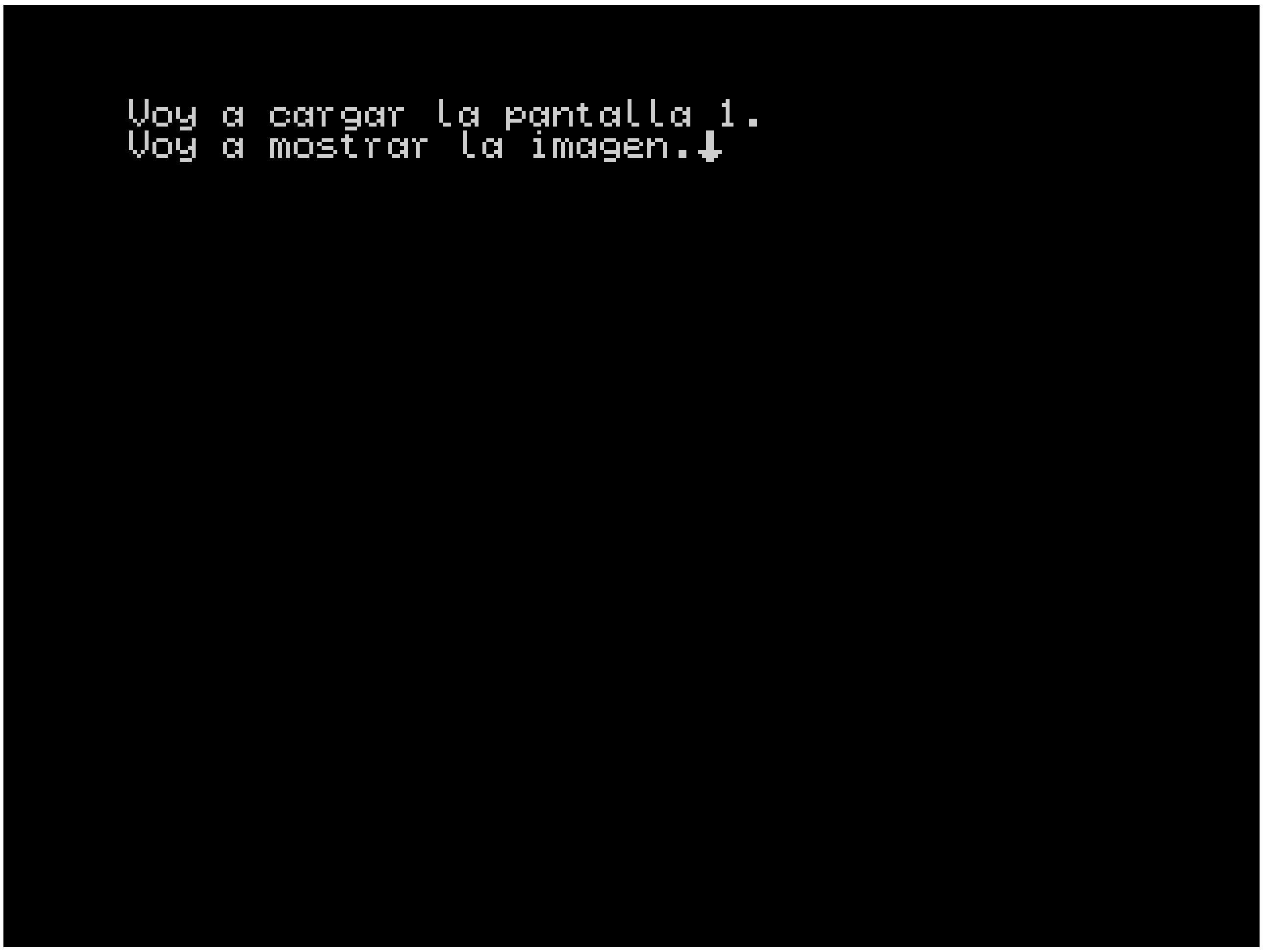
Before launching the emulator, look in the `\IMAGES` directory, where you will find `001.SCR` and `001.CSC`. The `MakeAdv` script looks for files named xxx.SCR, where the three x's are digits, and compresses them with `CSC`, generating the corresponding files with the extension `.CSC`. The engine will search the disk for files with this name when asked to load images.

Now we can launch the emulator. The first thing you will see is this:



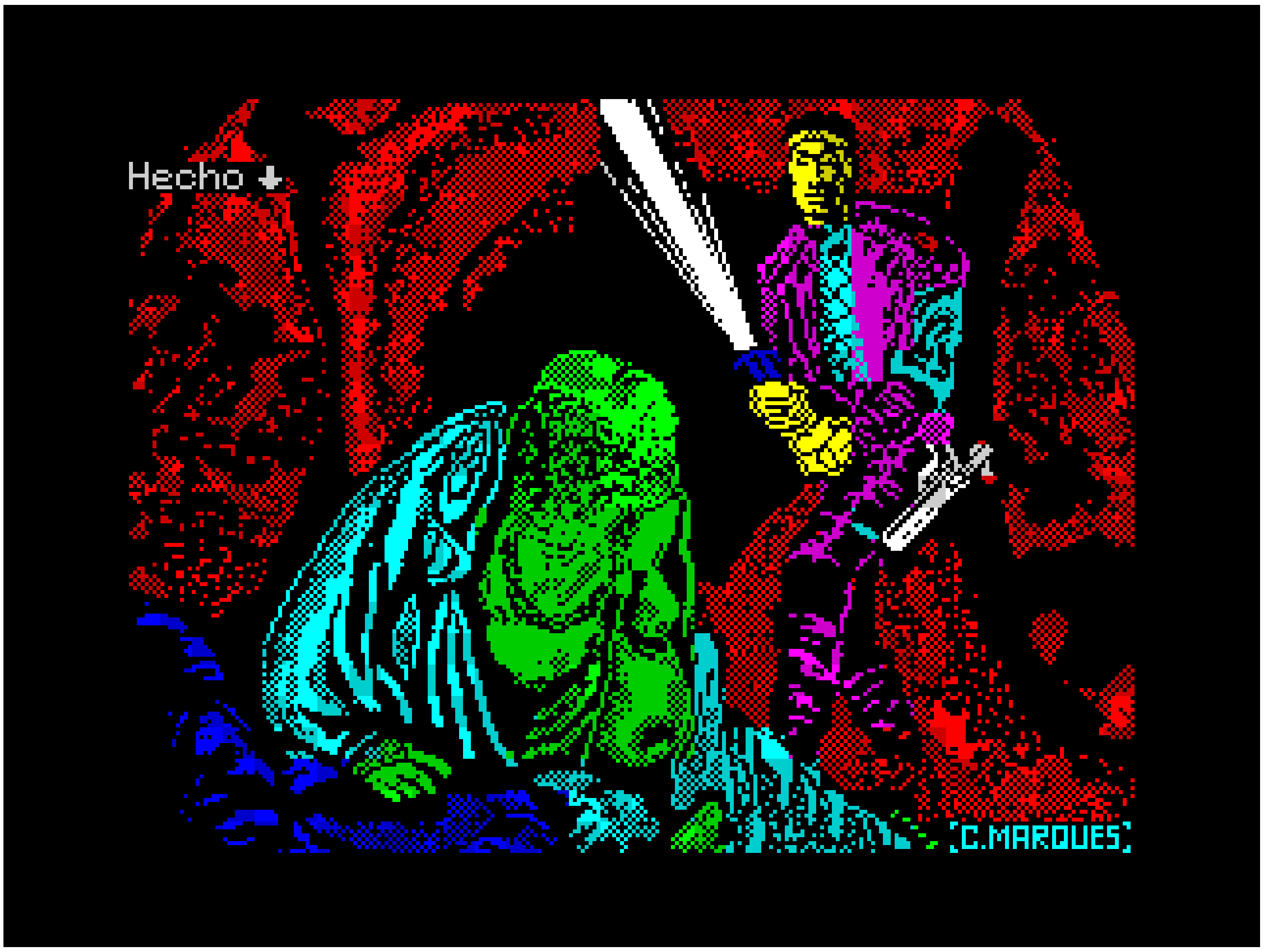
```
Voy a cargar la pantalla 1. ⬇
```

When you press the confirm key, the `PICTURE 1` command will be launched. What this command will do is load and decompress the image file `001.CSC` into memory, but be careful, it is not displayed yet! You will notice that the disk has been accessed (this depends on the emulator).



```
Voy a cargar la pantalla 1.
Voy a mostrar la imagen.
```

With this we have the image loaded, but to display it, we have to use the `DISPLAY 1` command, and then the image is displayed:



We can now display images, but there are a few things to clear up first. The first thing you might be wondering is... what is the 1 in DISPLAY for? As stated in the reference, the `DISPLAY` command needs a parameter indicating whether to display the image or not; if the value is zero, it does not display it, and if it is non-zero, it does. This may seem pointless, but it makes sense if used with variables, to cause the image to be displayed conditionally based on the value of a variable.

Another thing you might be wondering is why are the commands to load the image and display it separate, instead of using a single command to do both? Well, the answer is a design decision for the disk version, since by separating the loading into a separate operation, we can control when it is done to, for example, load it when a chapter starts, and then display the image at the most opportune moment, since loading will stop the engine and pause the reading at an undesirable moment.

For now, just remember that you first need `PICTURE 3`, to load the image `003.CSC`, for example, and then `DISPLAY 1` to display it. Note that we can only load one image at a time, so if we load another image, the one already loaded will be deleted, and a loaded image can be displayed as many times as we want. And you'll get a nice error if you try to load an image that doesn't exist on disk or in memory, or when displaying an image without loading it first.

When displaying images, keep in mind that whatever is already on the screen will always be overwritten. The default behavior is to load images full screen (192 lines), but you can edit the number of lines to load by modifying the value of the `IMGLINES` variable in the `make_adv.cmd` script:

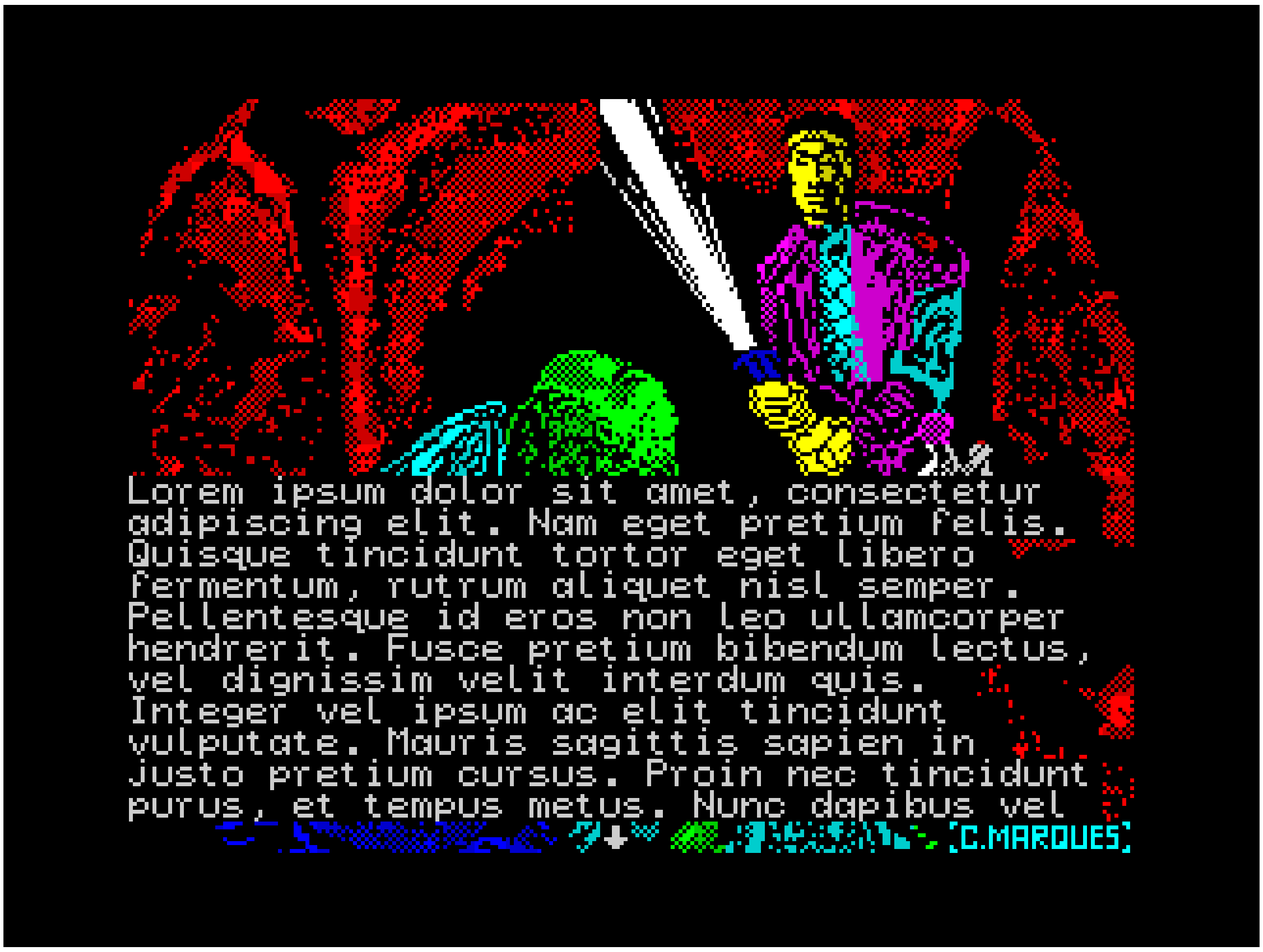
```
REM Number of lines used on SCR files at compressing
SET IMGLINES=192
```

Due to the color limitations of the Spectrum, I recommend that it always be a multiple of 8.

When using images, we can adjust the size of the print area so that it does not overwrite the entire drawing using `MARGINS`. Let's combine two previous examples to see this:

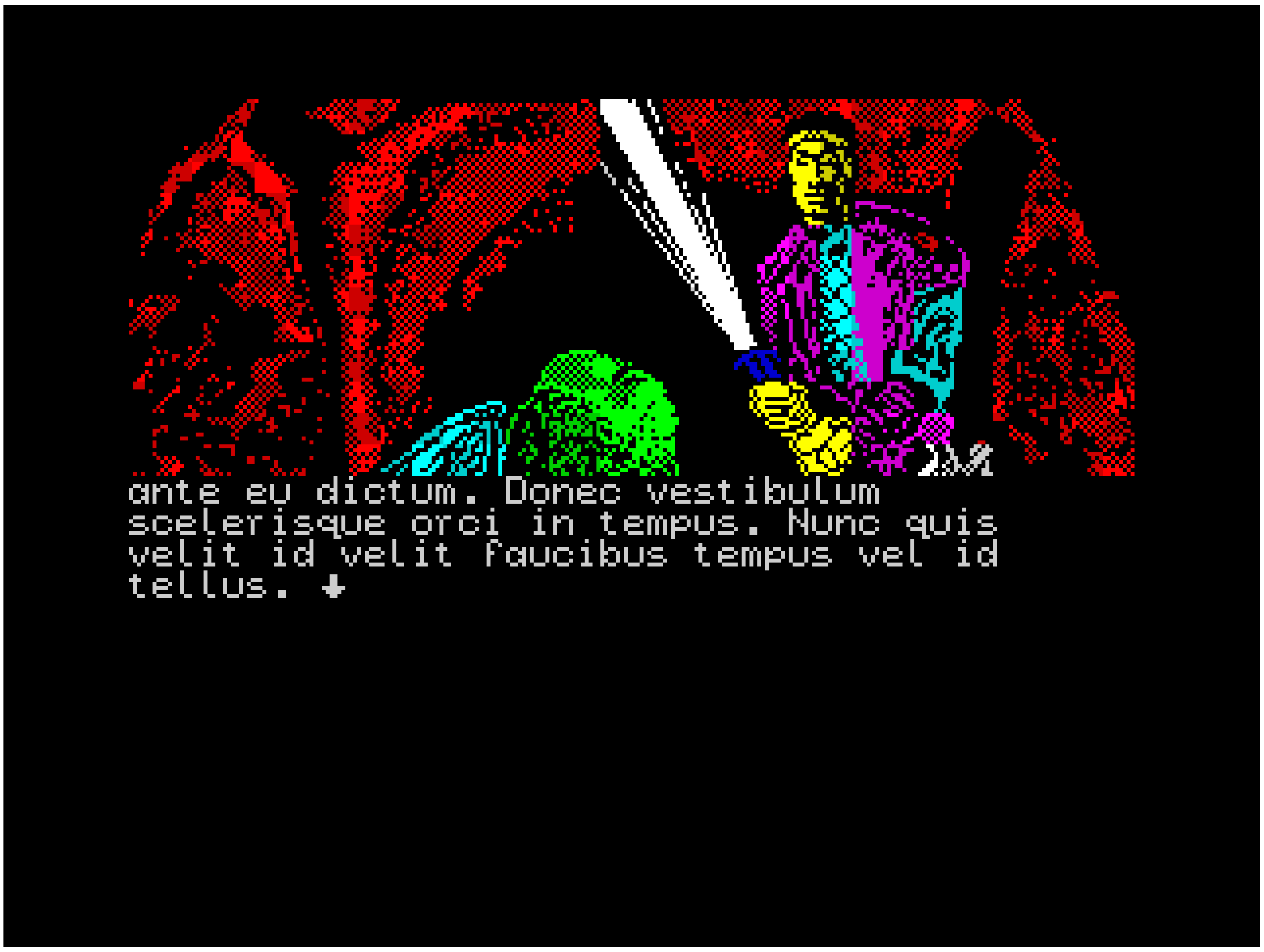
```
[[ /* Pone colores de pantalla y la borra */
   PAPER 0    /* Color de fondo negro  */
   INK   7    /* Color de texto blanco */
   BORDER 0   /* Borde de color negro  */
   CLEAR      /* Borramos la pantalla*/
   PAGEPAUSE 1
   PICTURE 1
   DISPLAY 1
   MARGINS 0, 10, 32, 14
]]Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nam eget pretium felis.
Quisque tincidunt tortor eget libero fermentum, rutrum aliquet nisl semper.
Pellentesque id eros non leo ullamcorper hendrerit. Fusce pretium bibendum lectus,
vel dignissim velit interdum quis. Integer vel ipsum ac elit tincidunt vulputate.
Mauris sagittis sapien in justo pretium cursus. Proin nec tincidunt purus, et
tempus metus. Nunc dapibus vel ante eu dictum. Donec vestibulum scelerisque orci
in tempus. Nunc quis velit id velit faucibus tempus vel id tellus.[[ WAITKEY: END
]]
```

The first thing it will do is display the image, which will overwrite the entire screen, and then we will define a lower area where the text will be drawn:



Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nam eget pretium felis. Quisque tincidunt tortor eget libero fermentum, rutrum aliquet nisl semper. Pellentesque id eros non leo ullamcorper hendrerit. Fusce pretium bibendum lectus, vel dignissim velit interdum quis. Integer vel ipsum ac elit tincidunt vulputate. Mauris sagittis sapien in justo pretium cursus. Proin nec tincidunt purus, et tempus metus. Nunc dapibus vel

Since the text does not fit, clicking continue will delete the printing area defined with `MARGINS` and continue printing the rest:

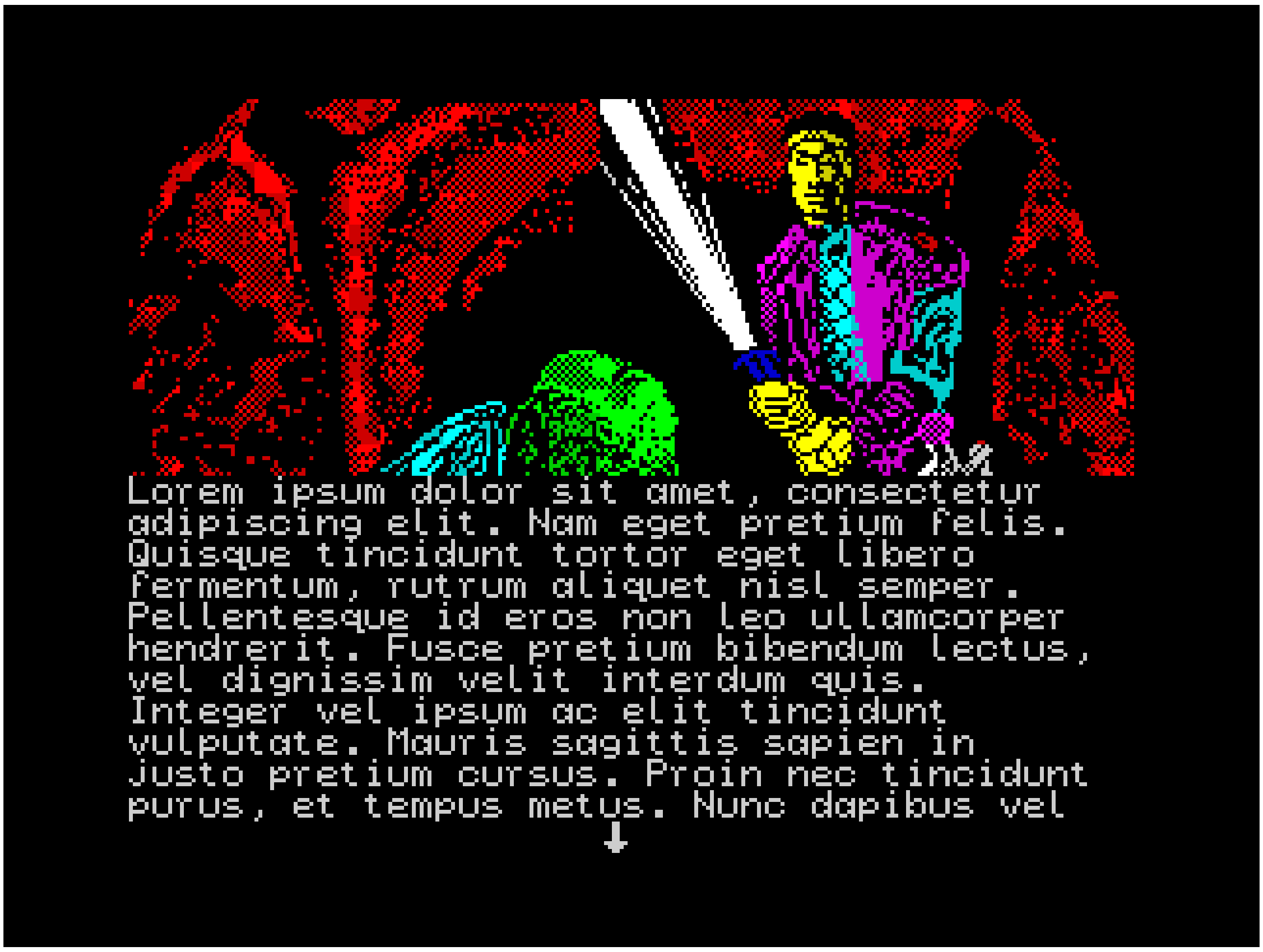


```
ante eu dictum. Donec vestibulum
scelerisque orci in tempus. Nunc quis
velit id velit faucibus tempus vel id
tellus. ↓
```

To make the behavior a little more consistent, let's put a `CLEAR` right after `MARGINS`:

```
[[ /* Pone colores de pantalla y la borra */
   PAPER 0    /* Color de fondo negro  */
   INK   7    /* Color de texto blanco */
   BORDER 0   /* Borde de color negro  */
   CLEAR      /* Borramos la pantalla*/
   PAGEPAUSE 1
   PICTURE 1
   DISPLAY 1
   MARGINS 0, 10, 32, 14
   CLEAR
]]Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nam eget pretium felis.
Quisque tincidunt tortor eget libero fermentum, rutrum aliquet nisl semper.
Pellentesque id eros non leo ullamcorper hendrerit. Fusce pretium bibendum lectus,
vel dignissim velit interdum quis. Integer vel ipsum ac elit tincidunt vulputate.
Mauris sagittis sapien in justo pretium cursus. Proin nec tincidunt purus, et
tempus metus. Nunc dapibus vel ante eu dictum. Donec vestibulum scelerisque orci
in tempus. Nunc quis velit id velit faucibus tempus vel id tellus.[[ WAITKEY: END
]]
```

And with this it's better:



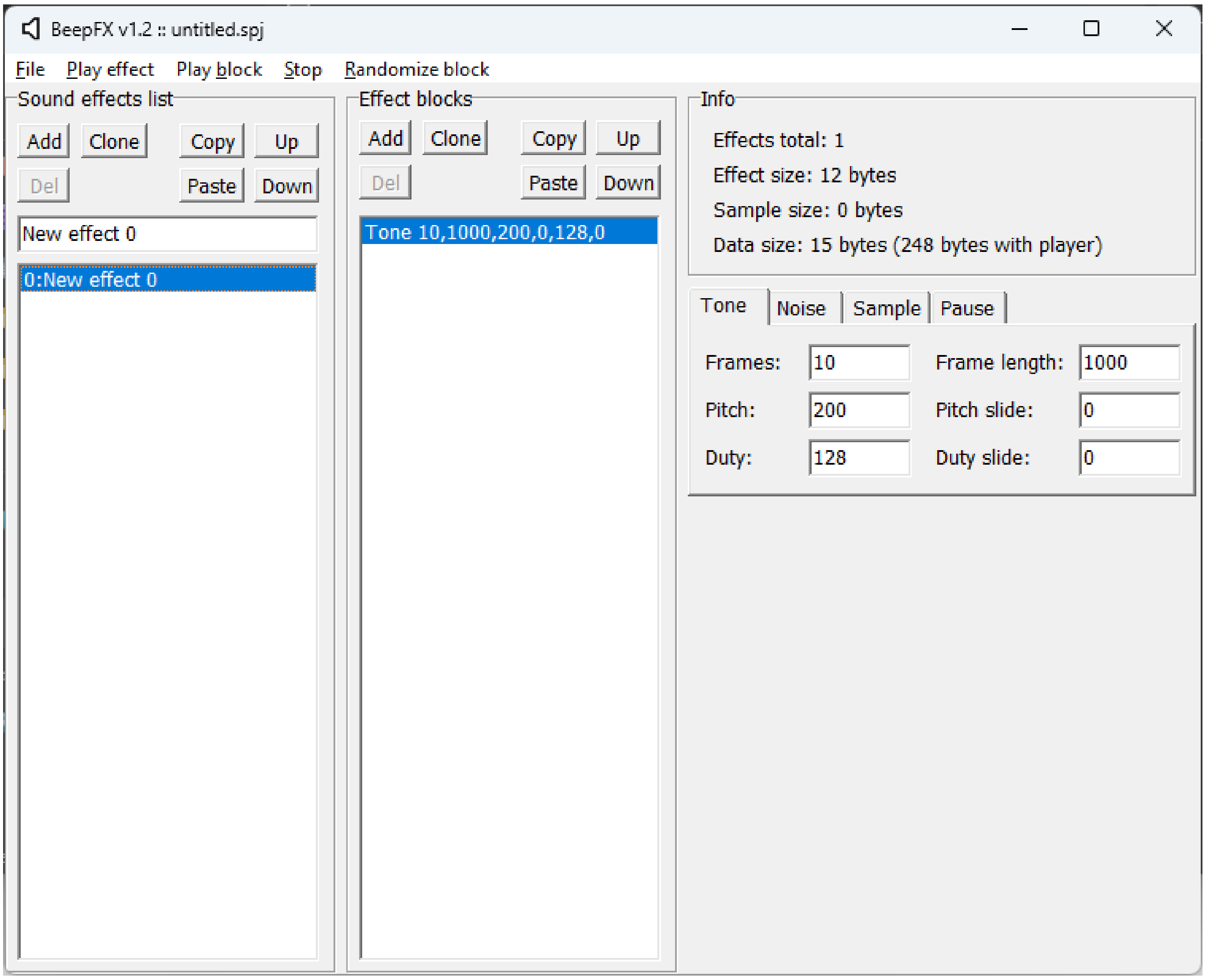
```
Lorem ipsum dolor sit amet, consectetur
adipiscing elit. Nam eget pretium felis.
Quisque tincidunt tortor eget libero
fermentum, rutrum aliquet nisl semper.
Pellentesque id eros non leo ullamcorper
hendrerit. Fusce pretium bibendum lectus,
vel dignissim velit interdum quis.
Integer vel ipsum ac elit tincidunt
vulputate. Mauris sagittis sapien in
justo pretium cursus. Proin nec tincidunt
purus, et tempus metus. Nunc dapibus vel
```

Here we've used a random image, more specifically the loading screen from the *La Adventura Original*. But if we're going to do this kind of cropping with all images, we can change the value of the `IMGLINES` variable, so that it crops out the lines we don't want. This will save space, and prevent the bottom text area from being overwritten when we load a new image.

For image management and taking into account these restrictions, it's important to plan in advance how and when we're going to display them.

Sound effects (Beeper)

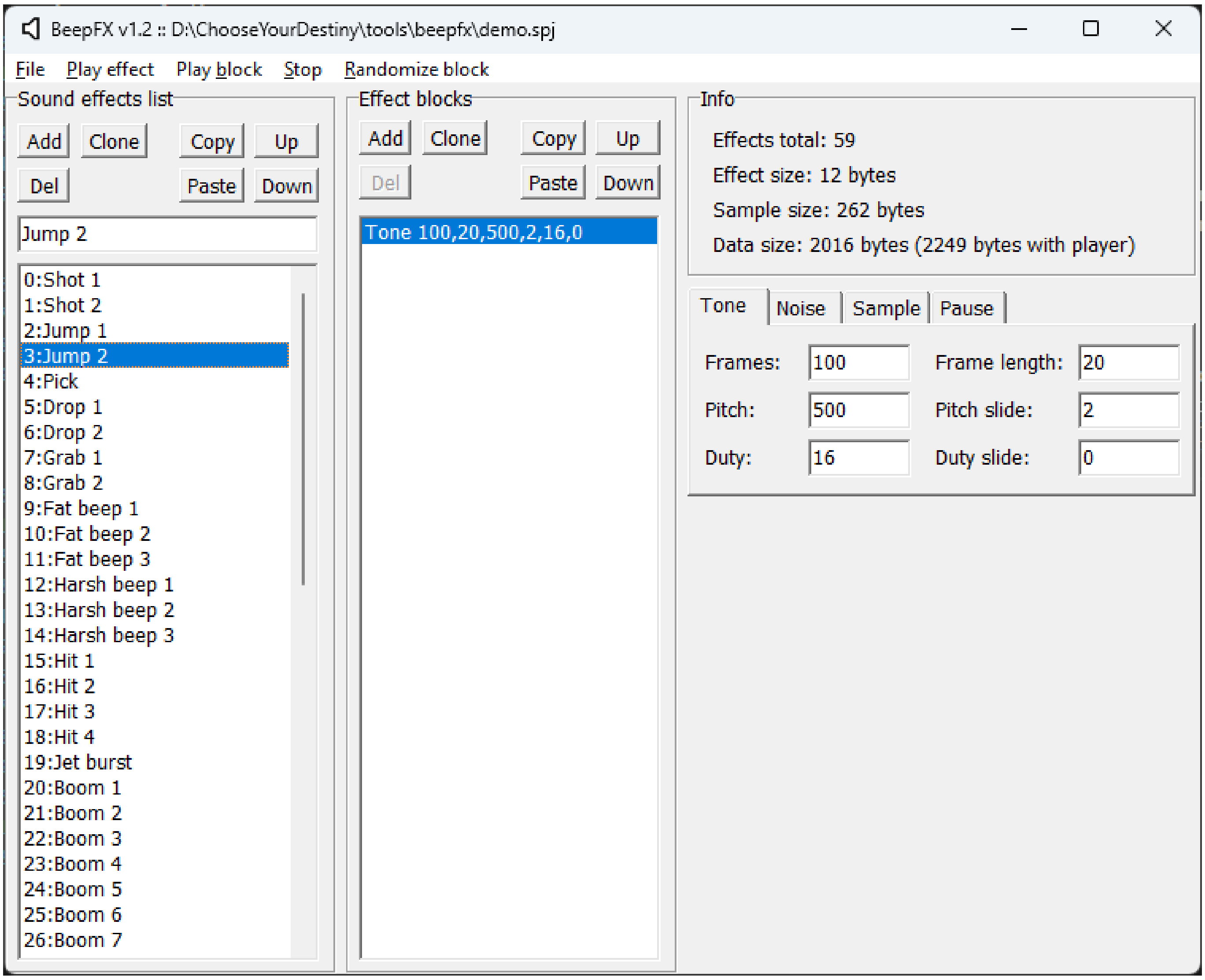
To enhance the atmosphere of our adventure, the engine allows us to play sound effects through the Beeper. To do this we use the tool BeepFx from Shiru, a tool widely used in new developments for Spectrum.



Effects total: 1
Effect size: 12 bytes
Sample size: 0 bytes
Data size: 15 bytes (248 bytes with player)

Frames: 10
Frame length: 1000
Pitch: 200
Pitch slide: 0
Duty: 128
Duty slide: 0

In this tutorial we are not going to teach how to use the tool, but we are going to take one of the example files included in the package. From the menu `File -> Open project`, we open the file `demo.spj`, where there are a series of effects already created. We can reproduce them with the option `Play Effect`:



BeepFX v1.2 :: D:\ChooseYourDestiny\tools\beepfx\demo.spj

Jump 2

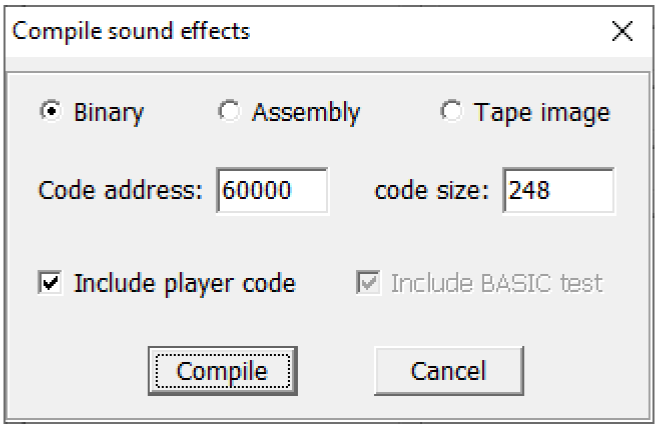
```
0:Shot 1
1:Shot 2
2:Jump 1
3:Jump 2
4:Pick
5:Drop 1
6:Drop 2
7:Grab 1
8:Grab 2
9:Fat beep 1
10:Fat beep 2
11:Fat beep 3
12:Harsh beep 1
13:Harsh beep 2
14:Harsh beep 3
15:Hit 1
16:Hit 2
17:Hit 3
18:Hit 4
19:Jet burst
20:Boom 1
21:Boom 2
22:Boom 3
23:Boom 4
24:Boom 5
25:Boom 6
26:Boom 7
```

Tone 100,20,500,2,16,0

Effects total: 59
Effect size: 12 bytes
Sample size: 262 bytes
Data size: 2016 bytes (2249 bytes with player)

Frames:	100	Frame length:	20
Pitch:	500	Pitch slide:	2
Duty:	16	Duty slide:	0

Now we are going to export it to a file with which we can use it with the engine. To do this, go to the menu `File -> Compile`, where this window will appear:

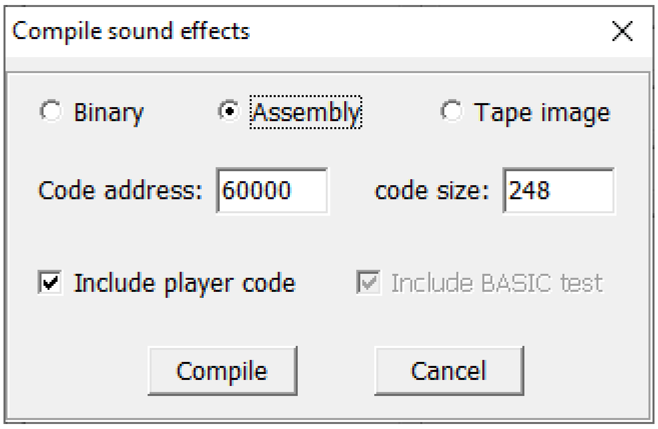


[✓ Binary] Assembly Tape image

Code address: 60000 code size: 248

[✓ Include player code] [✓ Include BASIC test]

Now comes the important part, **we must always leave the `Assembly` and `Include player code` options checked**:


Compile sound effects

Binary [✓ Assembly] Tape image

Code address: 60000 code size: 248

[✓ Include player code] [✓ Include BASIC test]

We click on the `Compile` button and a dialog box will appear to save the file. **We must call it `SFX.ASM`** and save it in the folder where we are developing our adventure. When we run the script `make_adv.cmd`, if it finds the file `SFX.ASM` in the same directory, it will automatically include it in the resulting file and we can use it from the engine.

Let's give an example, put this as the code for the adventure:

```
[[ /* Pone colores de pantalla y la borra */
   PAPER 0     /* Color de fondo negro   */
   BORDER 0    /* Borde de color negro   */
   INK   7     /* Color de texto blanco */
   PAGEPAUSE 1
   LABEL Menu
   CLEAR]]Selecciona el efecto a reproducir:

  [[ OPTION GOTO Efecto0 ]]Efecto 0
  [[ OPTION GOTO Efecto1 ]]Efecto 1
  [[ OPTION GOTO Efecto2 ]]Efecto 2
  [[ OPTION GOTO Efecto3 ]]Efecto 3
  [[ OPTION GOTO Efecto4 ]]Efecto 4
  [[ OPTION GOTO Final ]]Salir

[[ CHOOSE
   LABEL Efecto0 : SFX 0 : GOTO Menu
   LABEL Efecto1 : SFX 1 : GOTO Menu
   LABEL Efecto2 : SFX 2 : GOTO Menu
   LABEL Efecto3 : SFX 3 : GOTO Menu
   LABEL Efecto4 : SFX 4 : GOTO Menu
   LABEL Final]]Adios...[[WAITKEY: END ]]
```

As you can see, with the `SFX` command, we can play any of the effects in the file by specifying its number as a parameter.

A peculiarity of the SFX command is that, unlike images, if the SFX.BIN file is not on the disk, it will silently fail without giving an error and the sound effect will simply not be played.

Versions for the different Zx Spectrum models

So far, we have only developed adventures for the original 48k Zx Spectrum, however, the 128k models can also be played on tape or disk. The disk support has the peculiarity that the pictures and melodies are not stored in memory, but are loaded dynamically from disk when the `PICTURE` and `TRACK` commands are executed.

The available space will vary depending on the adventure and the extra resources to be used, since if music and sound effects are omitted, the size of the interpreter will be reduced, leaving more space for text and images. On average, you should expect to have about 96 Kb available on the 128K models and about 24 Kb on the 40K models. For this reason, the 48K version does not include support for AY melodies.

If you use the `make_adv.cmd` script, to change the model, you simply have to change the `TARGET` variable to the values `48k` or `128k` if you want to generate the TAPs for the corresponding models. For example, if we want to create a version of our tutorial for Spectrum 128k:

```
@echo off  &SETLOCAL

REM ---- Configuration variables

REM Name of the game
SET GAME=Tutorial
REM This name will be used as:
REM   - The file to compile will be test.txt with this example
REM   - The name of the TAP file or +3 disk image

REM Target for the compiler (48k, 128k for TAP, plus3 for DSK)
SET TARGET=128k

REM Number of lines used on SCR files at compressing
SET IMGLINES=192

REM Loading screen
SET LOAD_SCR=%~dp0\IMAGES\000.scr
```

With the `plus3` model, the output format will be a disk image in DSK format.

Music (AY)

CYD also allows playing music using the AY chip, using modules created with Vortex Tracker, in `PT3` format. AY music is not available for the 48K models, so we will need to change the `TARGET` variable in `made_adv.cmd`, as described in the previous chapter.

The operation is intentionally similar to that of the images. The module file names have to be 3-digit numbers with the extension `.PT3`, so that they are `000.PT3`, `001.PT3` and so on, and they must be included on the disk. To make things easier, the `make_adv.cmd` script will do it for us with all the files that follow this nomenclature and are inside the `.\TRACKS` folder.

The commands we have are also similar to the image management commands. With the `TRACK` command, we load a music module from disk into memory, so that with `TRACK 0`, we would load the module `000.PT3`, with

`TRACK 1` we would load `001.PT3`, etc.

Once the module is loaded, we would play it with the `PLAY 1` command when we need to do so. The parameter of the `PLAY` command is a number that, if it is zero, stops the music (if it is already playing), and if it is not zero, plays it from the beginning (if it is already stopped).

Finally, with the `LOOP` command we indicate whether we want the module to play only once or we want it to play indefinitely. If the value of its parameter is zero, it will only play once, and if it is non-zero, it will start playing again when it finishes.

One detail to keep in mind is that the effect of the `LOOP` command is specific to the built-in player and is only valid when the module can finish. The `.PT3` format has repeat commands, making the module play endlessly by itself, so it is convenient to play it with an external player to see if this is the case.

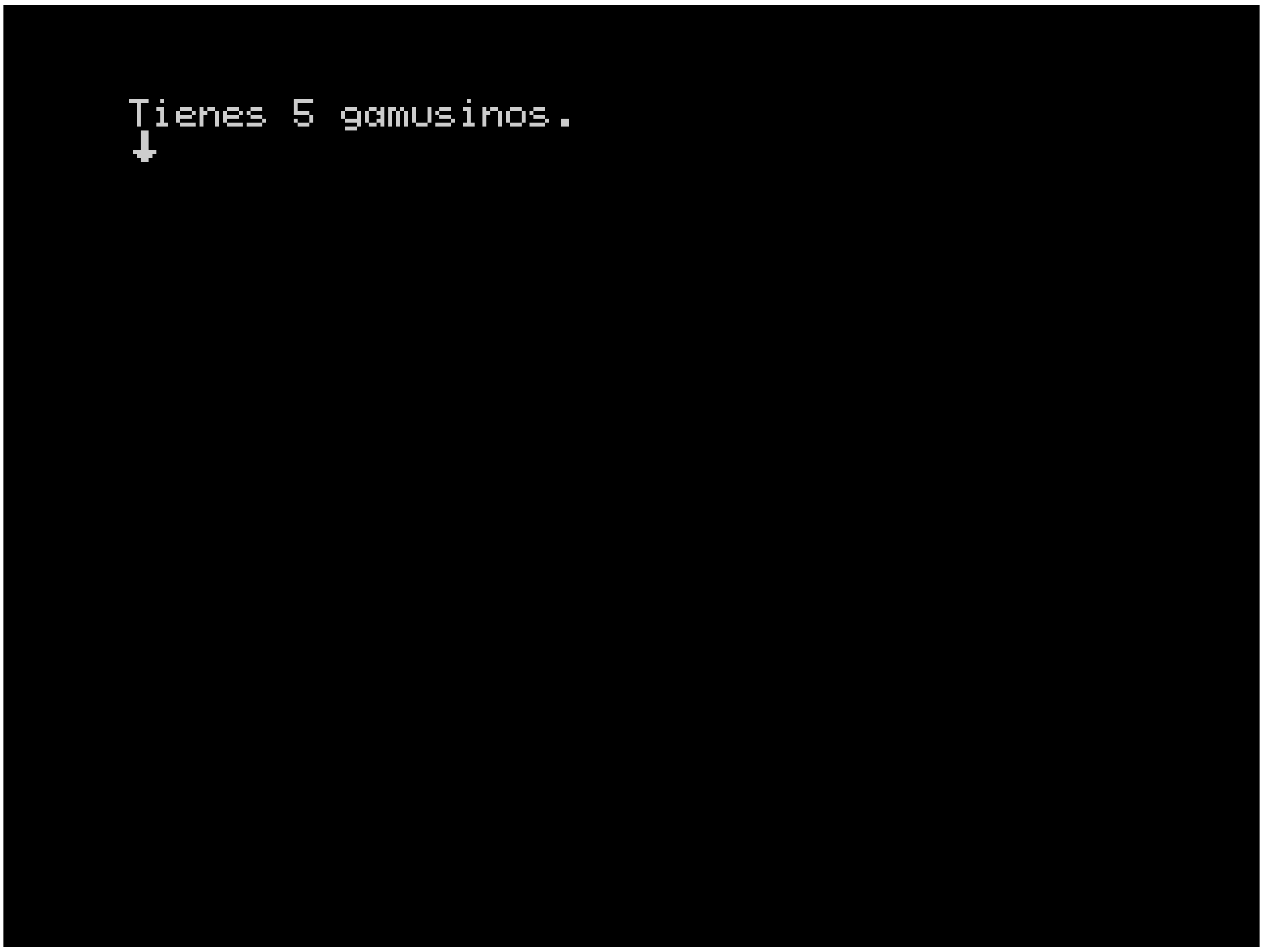
Variables

With what we already know, we could already make a relatively simple adventure based on options, like "Choose Your Own Adventure", but we can go further...

The engine has 256 variables or flags of a byte, that is, 256 warehouses where we can store values from 0 to 255. Each of these variables is identified in turn by a number from 0 to 255. Let's see it with an example:

```
[[ /* Pone colores de pantalla y la borra */
   PAPER 0    /* Color de fondo negro  */
   BORDER 0   /* Borde de color negro  */
   INK   7    /* Color de texto blanco */
   PAGEPAUSE 1
   CLEAR]][[SET 1 TO 5]]Tienes [[PRINT @1]] gamusinos.
[[WAITKEY : END]]
```

This is the result:

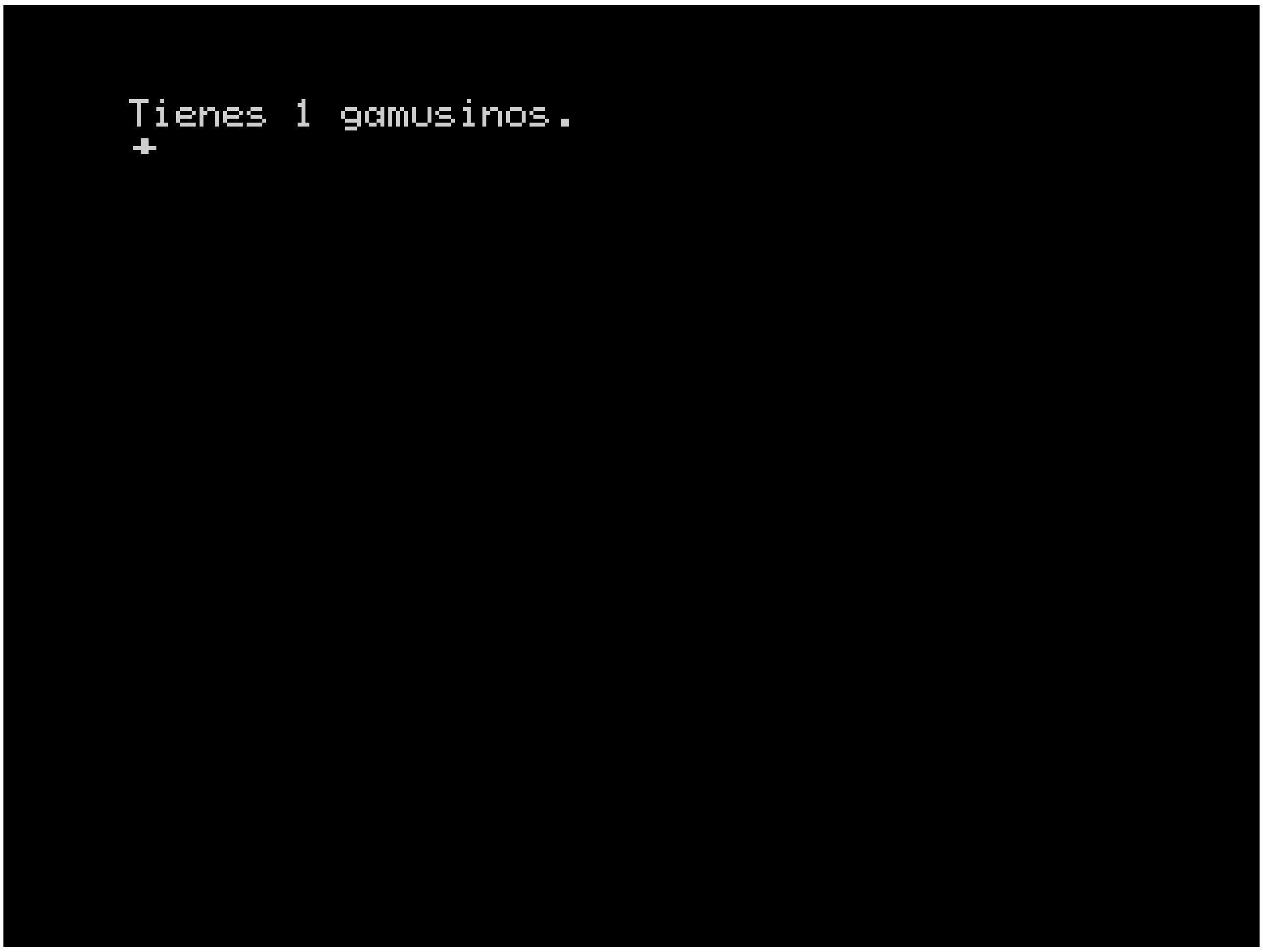


```
Tienes 5 gamusinos.
```

The first thing we see new is this `SET 1 TO 5`, with this we are telling the engine to save the value 5 inside the variable number 1. And as a consequence, with `PRINT @1` we are telling it to display the value of variable 1 on the screen.

One thing you might have noticed is that `PRINT` puts an at sign in front of the variable number. The at sign is the variable indicator. To explain it better, remove the at sign so that it looks like this:

```
[[ /* Pone colores de pantalla y la borra */
   PAPER 0    /* Color de fondo negro  */
   BORDER 0   /* Borde de color negro  */
   INK   7    /* Color de texto blanco */
   PAGEPAUSE 1
   CLEAR]][[SET 1 TO 5]]Tienes [[PRINT 1]] gamusinos.
[[WAITKEY : END]]
```



```
Tienes 1 gamusinos.
```

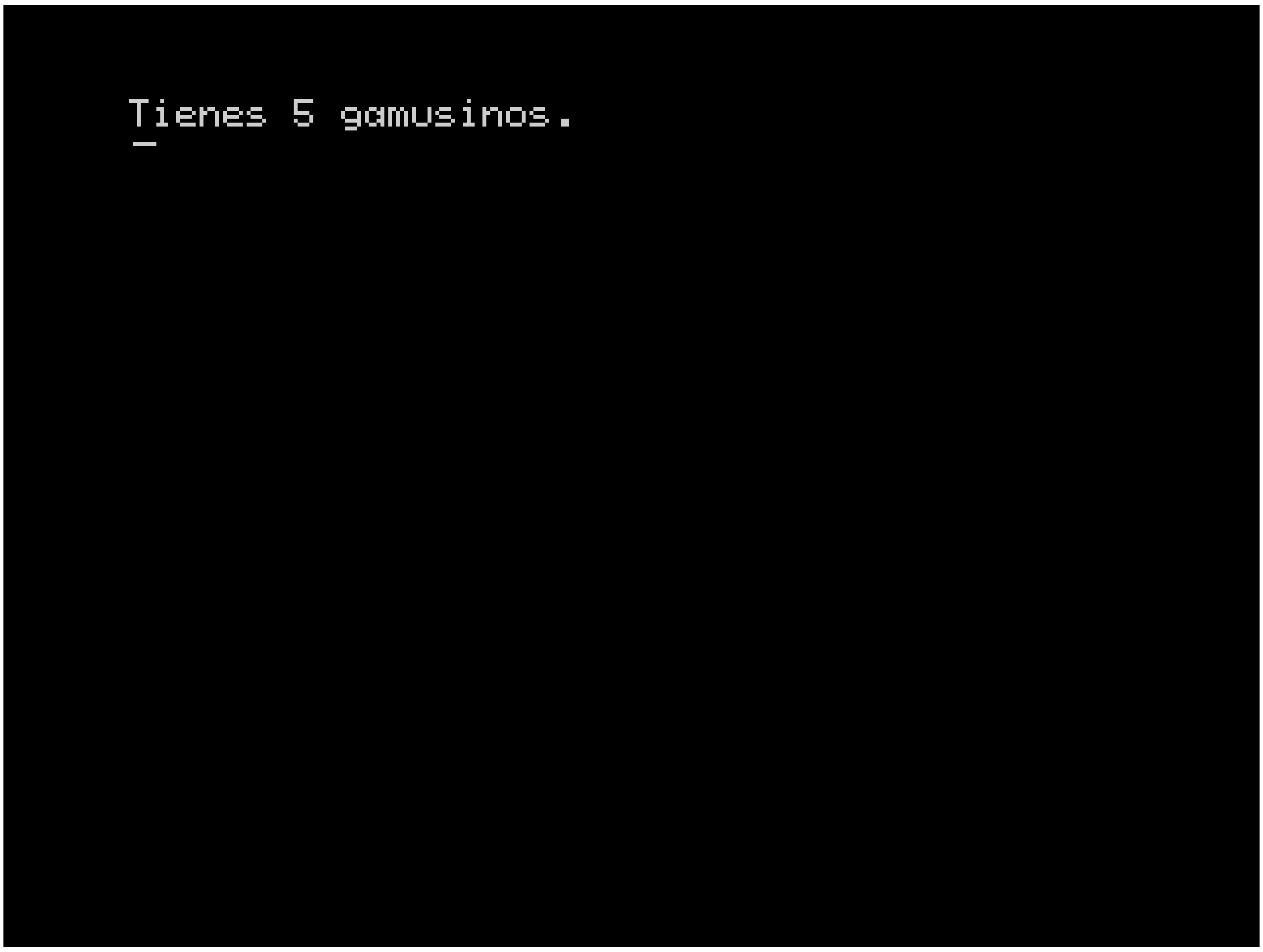
Wow! The thing is that when you put an @ sign in front, it means **take the value of the variable whose number I indicate after it**. If you have consulted the manual, you will have seen that almost all commands allow expressions with variables.

Therefore, with `PRINT 1`, what you are indicating is "Print the value 1", but with `PRINT @1`, what is indicated is "Print the content of the variable 1".

Let's reinforce this concept with this example:

```
[[ /* Pone colores de pantalla y la borra */
   PAPER 0    /* Color de fondo negro  */
   BORDER 0   /* Borde de color negro  */
   INK   7    /* Color de texto blanco */
   PAGEPAUSE 1
   CLEAR
   SET 1 TO 5
   SET 0 TO @1
]]Tienes [[PRINT @0]] gamusinos.
[[WAITKEY : END]]
```

Again, we have 5 gamusinos:


```
Tienes 5 gamusinos.
_
```

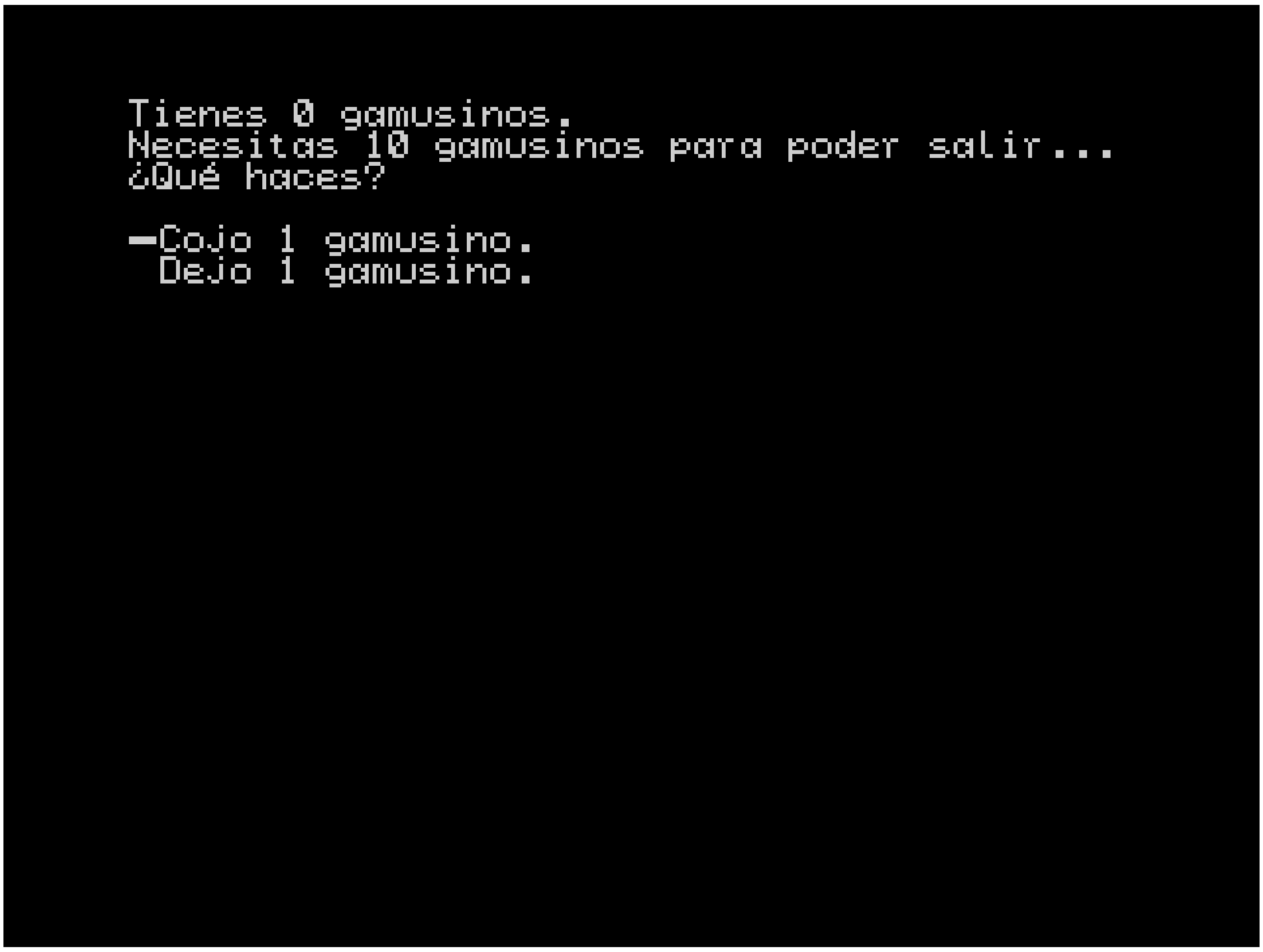
With `SET 1 TO 5` we store 5 in the variable num. 1. Then with `SET 0 TO @1`, what we do is store in the variable number 0 the value that variable number 1 contains and finally, with `PRINT @0`, we display the content of variable number 0.

Now we are going to see a more practical example of variables, which will test our knowledge acquired in this tutorial:

```
[[ /* Pone colores de pantalla y la borra */
   PAPER 0    /* Color de fondo negro  */
   BORDER 0   /* Borde de color negro  */
   INK   7    /* Color de texto blanco */
   PAGEPAUSE 1
   SET 0 TO 0
   #Inicio
   CLEAR
]]Tienes [[PRINT @0]] gamusinos.
Necesitas 10 gamusinos para poder salir...
¿Qué haces?

[[OPTION GOTO Suma1]]Cojo 1 gamusino.
[[OPTION GOTO Resta1]]Dejo 1 gamusino.
[[
   IF @0 <> 10 THEN GOTO Escoger
   OPTION GOTO Final]]Salir.
[[
   #Escoger
```

```
    CHOOSE
    #Suma1
    SET 0 TO @0 + 1
    GOTO Inicio
    #Resta1
    SET 0 TO @0 - 1
    GOTO Inicio
    #Final]]¡Gracias por jugar![[WAITKEY : END]]
```

It shows us this menu:

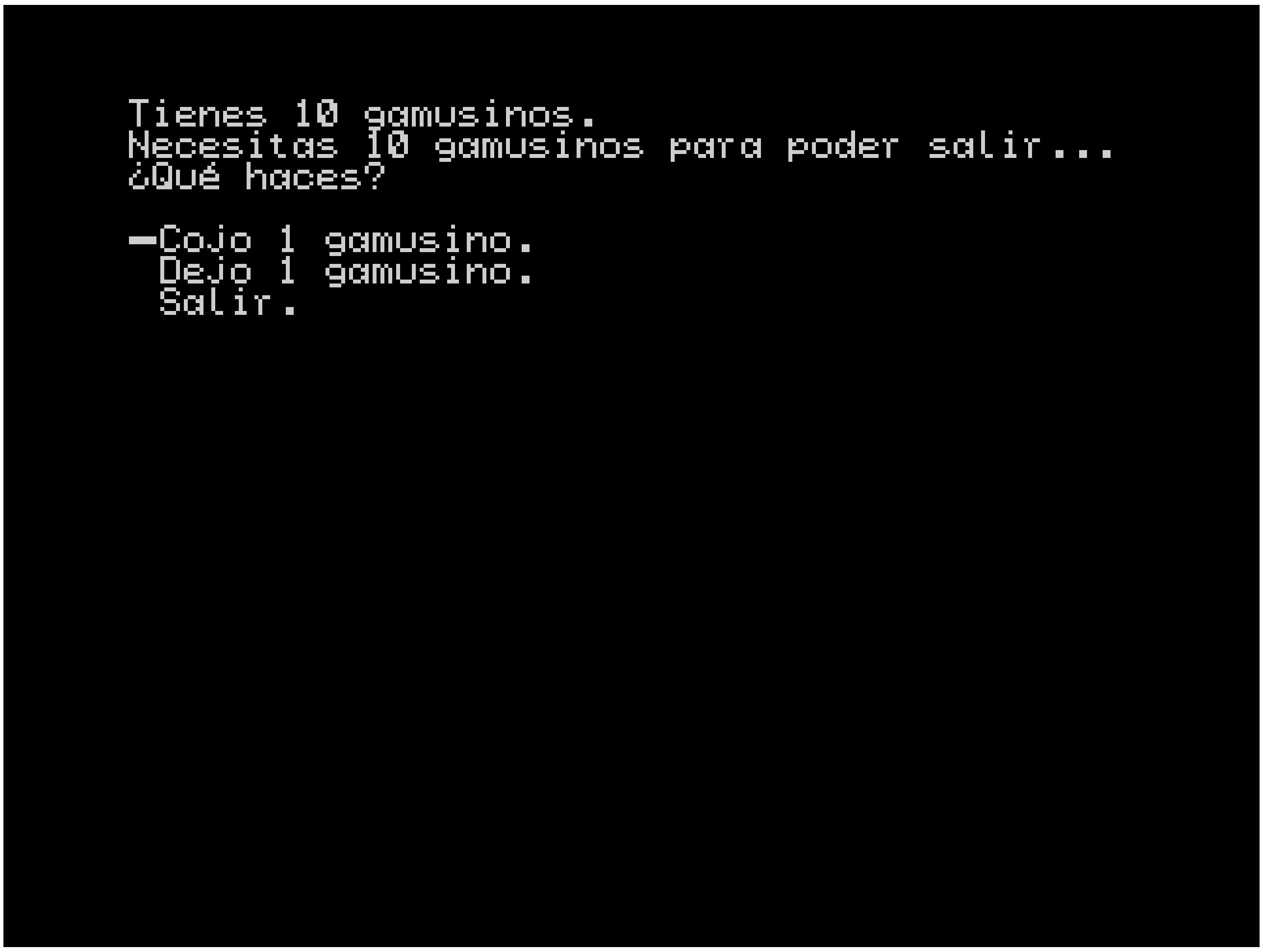


```
Tienes 0 gamusinos.
Necesitas 10 gamusinos para poder salir...
¿Qué haces?

-Cojo 1 gamusino.
 Dejo 1 gamusino.
```

If we choose the first option, it adds 1 to the variable 0, which is done by the command `SET 0 TO @0 + 1`, and the second option subtracts, and it does so with `SET 0 TO @0 - 1`.

The first thing that may catch your attention is that if I hit subtract when the value is zero, it does nothing. This is correct, you cannot subtract below zero or add above 255 in the variables.

And the second, where is the exit option? We are going to catch gamusinos until we have 10, as indicated:



```
Tienes 10 gamusinos.
Necesitas 10 gamusinos para poder salir...
¿Qué haces?

-Cojo 1 gamusino.
 Dejo 1 gamusino.
 Salir.
```

Now we can exit! The secret is in the conditionals. If we look at the exit option, we see that before it there is `IF 0 <> 10 THEN GOTO Choose`, which means "If the variable zero is not equal to 10, jump to the 'Choose' label", which causes the "Exit" option to be skipped and not reflected in the menu until the value of the variable 0 is 10.

That is, with variables and conditions, we have the necessary tools to make menus of options or texts that vary depending on certain conditions and make our adventure more dynamic.

Variable declaration

Looking at this program, it can seem quite cryptic:

```
[[ /* Pone colores de pantalla y la borra */
   PAPER 0    /* Color de fondo negro  */
   BORDER 0   /* Borde de color negro  */
   INK   7    /* Color de texto blanco */
   PAGEPAUSE 1
   SET 0 TO 0
   #Inicio
   CLEAR
]]Tienes [[PRINT @0]] gamusinos.
Necesitas 10 gamusinos para poder salir...
¿Qué haces?

[[OPTION GOTO Suma1]]Cojo 1 gamusino.
```

```
[[OPTION GOTO Resta1]]Dejo 1 gamusino.
[[
   IF 0 <> 10 THEN GOTO Escoger
   OPTION GOTO Final]]Salir.
[[
   #Escoger
   CHOOSE
   #Suma1
   SET 0 TO @0 + 1
   GOTO Inicio
   #Resta1
   SET 0 TO @0 - 1
   GOTO Inicio
   #Final]]¡Gracias por jugar![[WAITKEY : END]]
```

The first thing we are going to do is declare a variable, that is, give a name or alias to one of the flags so we can refer to it by its name. Let's take a look at it:

```
[[ /* Pone colores de pantalla y la borra */
   PAPER 0    /* Color de fondo negro  */
   BORDER 0   /* Borde de color negro  */
   INK   7    /* Color de texto blanco */
   PAGEPAUSE 1
   DECLARE 0 AS NumGamusinos
   SET NumGamusinos TO 0
   #Inicio
   CLEAR
]]Tienes [[PRINT @NumGamusinos]] gamusinos.
Necesitas 10 gamusinos para poder salir...
¿Qué haces?

[[OPTION GOTO Suma1]]Cojo 1 gamusino.
[[OPTION GOTO Resta1]]Dejo 1 gamusino.
[[
   IF 0 <> 10 THEN GOTO Escoger
   OPTION GOTO Final]]Salir.
[[
   #Escoger
   CHOOSE
   #Suma1
   SET NumGamusinos TO @NumGamusinos + 1
   GOTO Inicio
   #Resta1
   SET NumGamusinos TO @NumGamusinos - 1
   GOTO Inicio
   #Final]]¡Gracias por jugar![[WAITKEY : END]]
```

This is much better, with `DECLARE 0 AS NumGamusinos` we are telling the compiler that, from now on, the variable 0 will be called NumGamusinos, and we can use that name instead. With this we can give a meaningful name to the flags. Also keep in mind the following:

- We can still refer to the variable by its number.
- We cannot declare a variable and a label with the same name.
- We cannot declare two different variables with the same name.
- We can give two different names to the same variable.

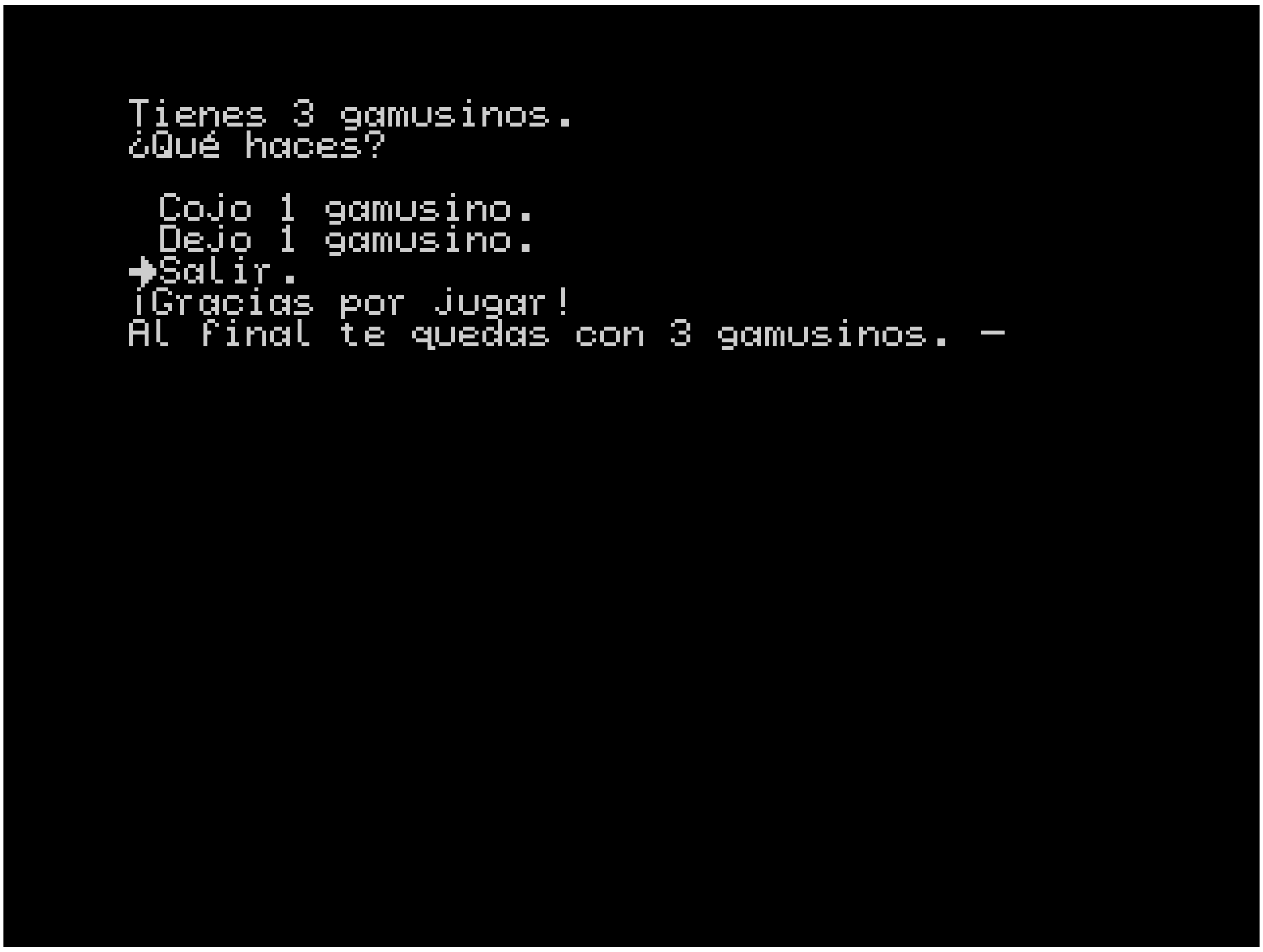
Subroutines

Subroutines are a special type of jump, which stores the position from which it was called, and which can be recovered later, allowing you to continue from the same point where you entered. This is a classic programming concept of "subroutines" or "subprograms", to perform repetitive tasks and which surely sounds familiar to those who have programmed something in Basic.

As always, let's see it with a simple example to understand it:

```
[[ /* Pone colores de pantalla y la borra */
   PAPER 0    /* Color de fondo negro  */
   BORDER 0   /* Borde de color negro  */
   INK   7    /* Color de texto blanco */
   PAGEPAUSE 1
   DECLARE 0 AS NumGamusinos
   SET NumGamusinos TO 0
   #Inicio
   CLEAR]]Tienes [[GOSUB ImprimirGamusimos]].
¿Qué haces?

[[OPTION GOTO Suma1]]Cojo 1 gamusino.
[[OPTION GOTO Resta1]]Dejo 1 gamusino.
[[OPTION GOTO Final]]Salir.
[[
   #Escoger
   CHOOSE
   #Suma1
   SET NumGamusinos TO @NumGamusinos + 1
   GOTO Inicio
   #Resta1
   SET NumGamusinos TO @NumGamusinos - 1
   GOTO Inicio
   #Final]]¡Gracias por jugar!
Al final te quedas con [[GOSUB ImprimirGamusimos]].[[
   WAITKEY
   END
   /* Subrutina de impresión de gamusinos*/
   #ImprimirGamusimos : PRINT @NumGamusinos]] gamusinos[[RETURN]]
```

The Gamusinos example again... But this time we print the number of Gamusinos twice, when we choose an option and at the very end.



```
Tienes 3 gamusinos.
¿Qué haces?

  Cojo 1 gamusino.
  Dejo 1 gamusino.
➔Salir.
¡Gracias por jugar!
Al final te quedas con 3 gamusinos. _
```

To do this we make a subroutine that performs this function:

```
[[LABEL ImprimirGamusimos]]Tienes [[PRINT @0]] gamusinos[[RETURN]]
```

We declare a label called `PrintGamusinos` that serves as a jump point. Then we have the printout of the number of gamusinos, and then the `RETURN` command. We make the call with `GOSUB PrintGamusinos` at the two points where we want it to be executed.

As I have already indicated, `GOSUB PrintGamusinos`, what it does is make a jump to the label `PrintGamusinos`, but with the exception that the point of the call to the subroutine is saved in the *stack*. Every subroutine must have a return point, that is, a completion point with which the engine is told to return to the point where we had left it, and that is done by the `RETURN` command, which recovers the last return point from the stack and jumps to that position.

One thing to note is that the subroutine is at the end, after `END`. This is so that it is not executed without us calling it explicitly. Please note that the interpreter does not differentiate a subroutine from "normal" code, and if it reaches that point it will execute it and do an invalid `RETURN` at the end. Therefore, I recommend to avoid these situations, placing them at the end after an `END`, or using a `GOTO` before it to skip it in case you accidentally reach it.

And now, as usual, the clarifications and exceptions. Subroutines can be nested, that is, you can call a subroutine inside another. The return addresses will be stored in the stack in reverse order, but **BE CAREFUL,**

the stack has a limit. This means you have a nesting limit. And as I explained in the previous paragraph, if you do a `RETURN` without a previous `GOSUB`, you will have at least one error, and at most, erroneous behavior.

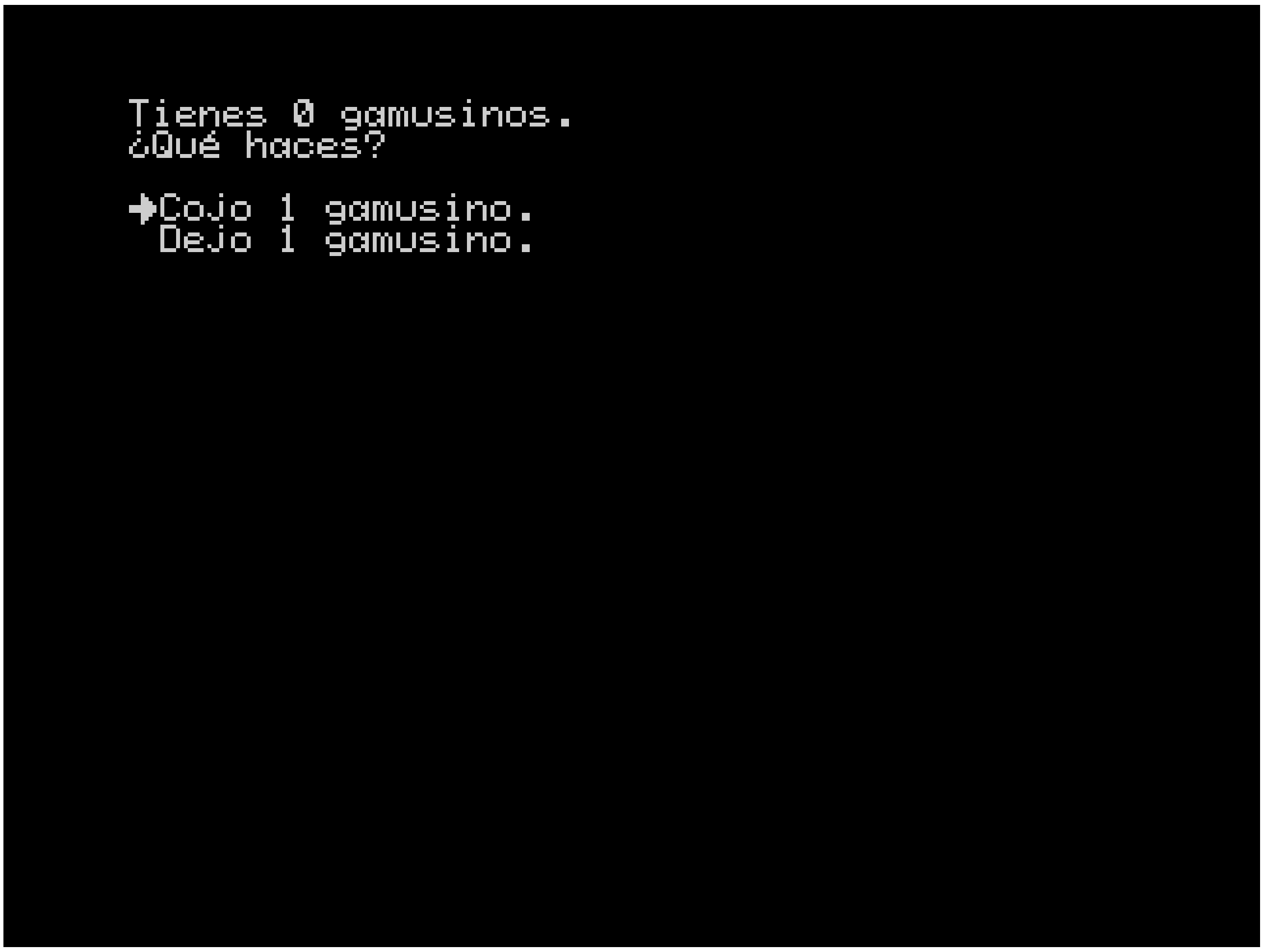
Generally, the ideal time to call a subroutine is when you make a choice from a menu. So we've included the `OPTION GOSUB Label` variant, which will call the corresponding subroutine if that option is chosen. When it reaches the `RETURN` (always remember to end subroutines with it!), it will resume execution right after the `CHOOSE` of that option. With our new knowledge, let's fine-tune our code:

```
[[ /* Pone colores de pantalla y la borra */
   PAPER 0    /* Color de fondo negro  */
   BORDER 0   /* Borde de color negro  */
   INK   7    /* Color de texto blanco */
   PAGEPAUSE 1
   DECLARE 0 AS NumGamusinos
   SET NumGamusinos TO 0
   #Inicio
   CLEAR]]Tienes [[GOSUB ImprimirGamusimos]].
¿Qué haces?

[[OPTION GOSUB Suma1]]Cojo 1 gamusino.
[[OPTION GOSUB Resta1]]Dejo 1 gamusino.
[[OPTION GOTO Final]]Salir.
[[
   CHOOSE
   GOTO Inicio
   #Final]]¡Gracias por jugar!
Al final te quedas con [[GOSUB ImprimirGamusimos]].[[
   WAITKEY
   END
   /* Subrutina de impresión de gamusinos*/
   #ImprimirGamusimos : PRINT @NumGamusinos]] gamusinos[[RETURN
   /* Subrutina de suma*/
   #Suma1 : SET NumGamusinos TO @NumGamusinos + 1 : RETURN
   /* Subrutina de resta */
   #Resta1 : SET NumGamusinos TO @NumGamusinos - 1 : RETURN
   ]]
```

Conditional Execution

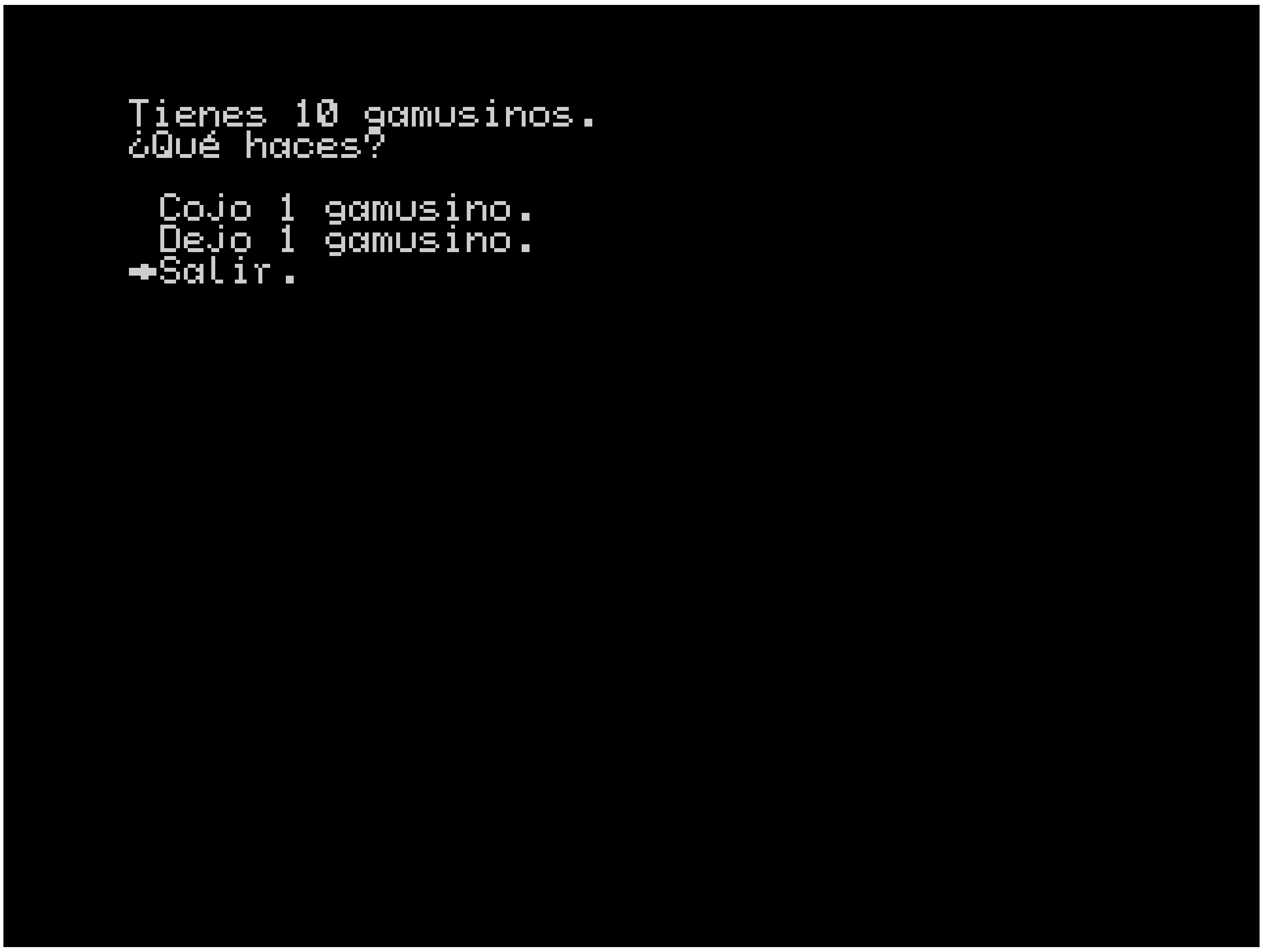
In the following example we will use the `IF ... THEN ... ENDIF` structure to illustrate how it works: Copy the following into `Tutorial.cyd`:

```
[[ /* Pone colores de pantalla y la borra */
   PAPER 0    /* Color de fondo negro  */
   BORDER 0   /* Borde de color negro  */
   INK   7    /* Color de texto blanco */
   PAGEPAUSE 1
   DECLARE 0 AS NumGamusinos
   SET NumGamusinos TO 0
   #Inicio
   CLEAR]]Tienes [[GOSUB ImprimirGamusimos]].
¿Qué haces?

[[OPTION GOSUB Suma1]]Cojo 1 gamusino.
[[OPTION GOSUB Resta1]]Dejo 1 gamusino.
[[ IF @NumGamusinos = 10 THEN
   OPTION GOTO Final]]Salir.
[[
   ENDIF
   CHOOSE
   GOTO Inicio
   #Final]]¡Gracias por jugar!
Al final te quedas con [[GOSUB ImprimirGamusimos]].[[
   WAITKEY
   END
   /* Subrutina de impresión de gamusinos*/
   #ImprimirGamusimos : PRINT @NumGamusinos]] gamusinos[[RETURN
   /* Subrutina de suma*/
   #Suma1 : SET NumGamusinos TO @NumGamusinos + 1 : RETURN
   /* Subrutina de resta */
   #Resta1 : SET NumGamusinos TO @NumGamusinos - 1 : RETURN
   ]]
```



```
Tienes 0 gamusinos.
¿Qué haces?

➜Cojo 1 gamusino.
 Dejo 1 gamusino.
```

Funny... now we can't get out...



```
Tienes 10 gamusinos.
¿Qué haces?

  Cojo 1 gamusino.
  Dejo 1 gamusino.
➡Salir.
```

...until we have 10 gamusinos! What happens?

The answer is the `IF @NumGamusinos = 10 THEN ... ENDIF`, if the condition that we have 10 gamusinos is met, whatever is between the `THEN` and the `ENDIF` is executed, and otherwise it will skip it.

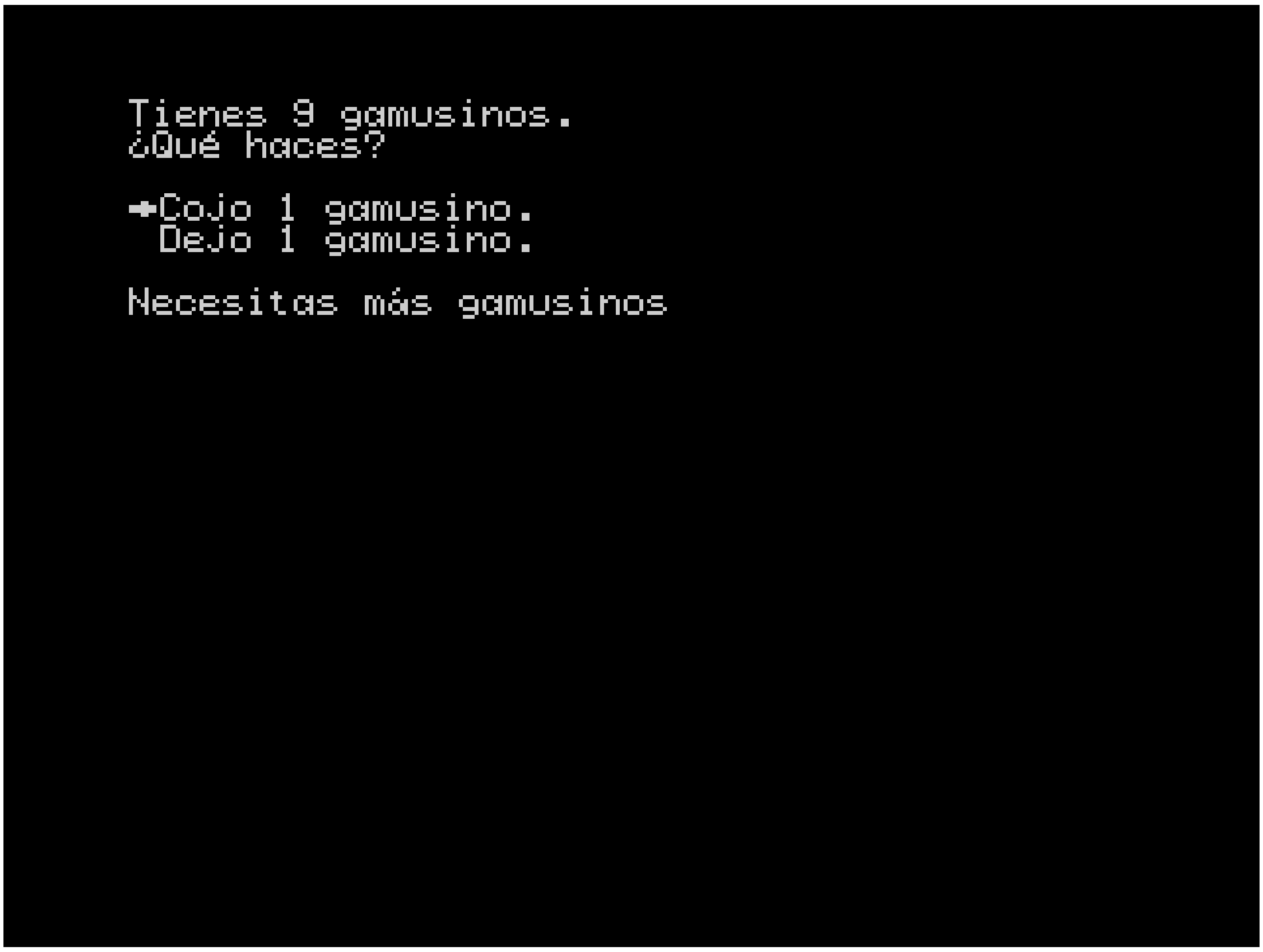
Note that not only the option command to exit is skipped; it also skips the text after it and, therefore, it does not show it either.

Now I am going to change to `IF @NumGamusinos = 10 THEN ... ELSE ... ENDIF`. What we do is add another block of code, which is executed if the condition is *NOT* met. In this case we are going to add another IF that tells us what we have to do:

```
[[ /* Pone colores de pantalla y la borra */
   PAPER 0    /* Color de fondo negro  */
   BORDER 0   /* Borde de color negro  */
   INK   7    /* Color de texto blanco */
   PAGEPAUSE 1
   DECLARE 0 AS NumGamusinos
   SET NumGamusinos TO 0
   #Inicio
   CLEAR]]Tienes [[GOSUB ImprimirGamusimos]].
¿Qué haces?

[[OPTION GOSUB Suma1]]Cojo 1 gamusino.
[[OPTION GOSUB Resta1]]Dejo 1 gamusino.
[[ IF @NumGamusinos = 10 THEN
```

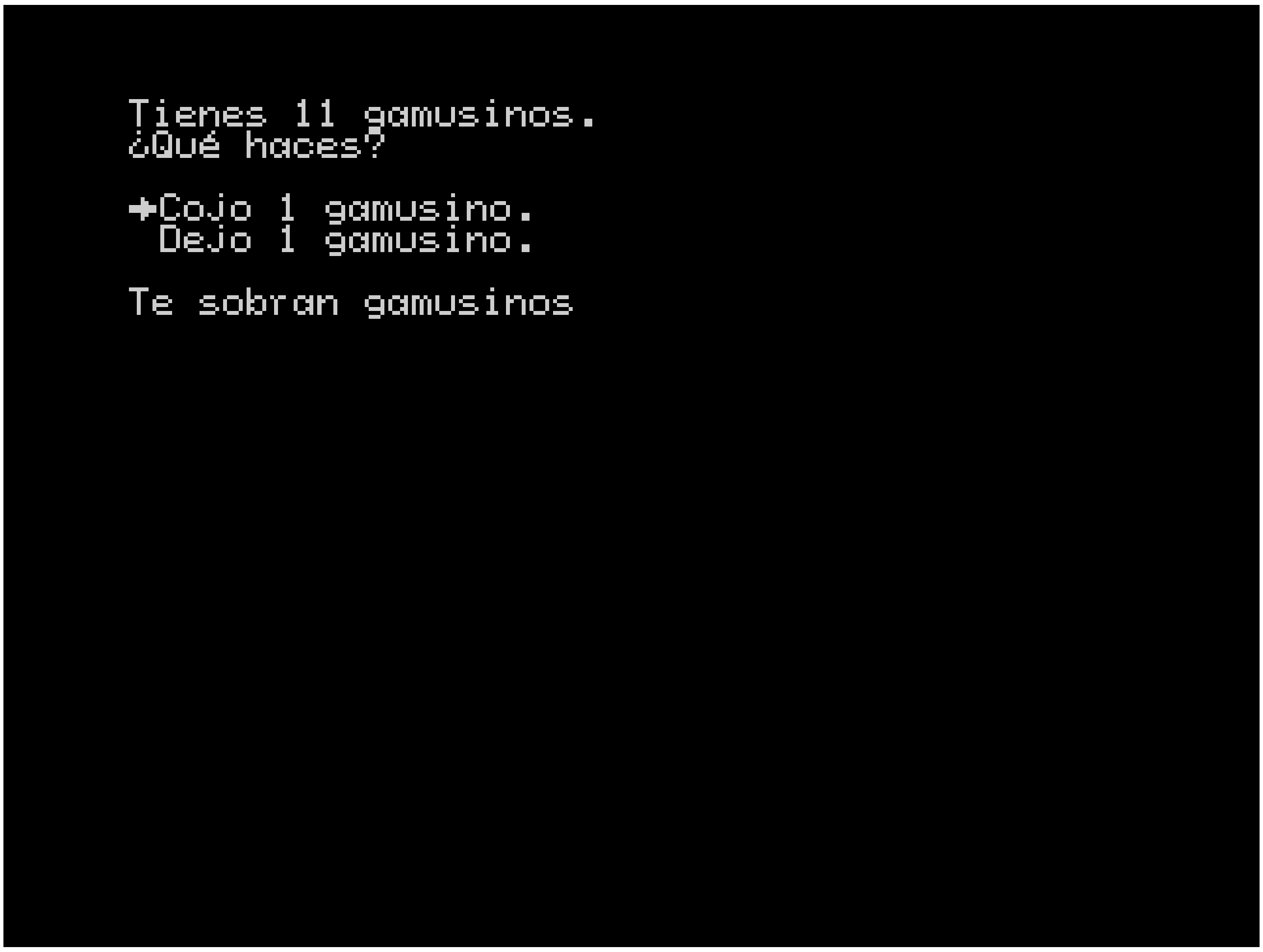
```
   OPTION GOTO Final]]Salir.
[[ ELSE IF @NumGamusinos < 10 THEN ]]
Necesitas más gamusinos
[[ ELSE ]]
Te sobran gamusinos
[[ ENDIF
   ENDIF
   CHOOSE
   GOTO Inicio
   #Final]]¡Gracias por jugar!
Al final te quedas con [[GOSUB ImprimirGamusimos]].[[
   WAITKEY
   END
   /* Subrutina de impresión de gamusinos*/
   #ImprimirGamusimos : PRINT @NumGamusinos]] gamusinos[[RETURN
   /* Subrutina de suma*/
   #Suma1 : SET NumGamusinos TO @NumGamusinos + 1 : RETURN
   /* Subrutina de resta */
   #Resta1 : SET NumGamusinos TO @NumGamusinos - 1 : RETURN
   ]]
```

If we are below:



```
Tienes 9 gamusinos.
¿Qué haces?

➡Cojo 1 gamusino.
 Dejo 1 gamusino.

Necesitas más gamusinos
```

And if we are above:



```
Tienes 11 gamusinos.
¿Qué haces?

➜Cojo 1 gamusino.
 Dejo 1 gamusino.

Te sobran gamusinos
```

This allows us, along with `GOTO` and `GOSUB`, to control the **flow of the program**.

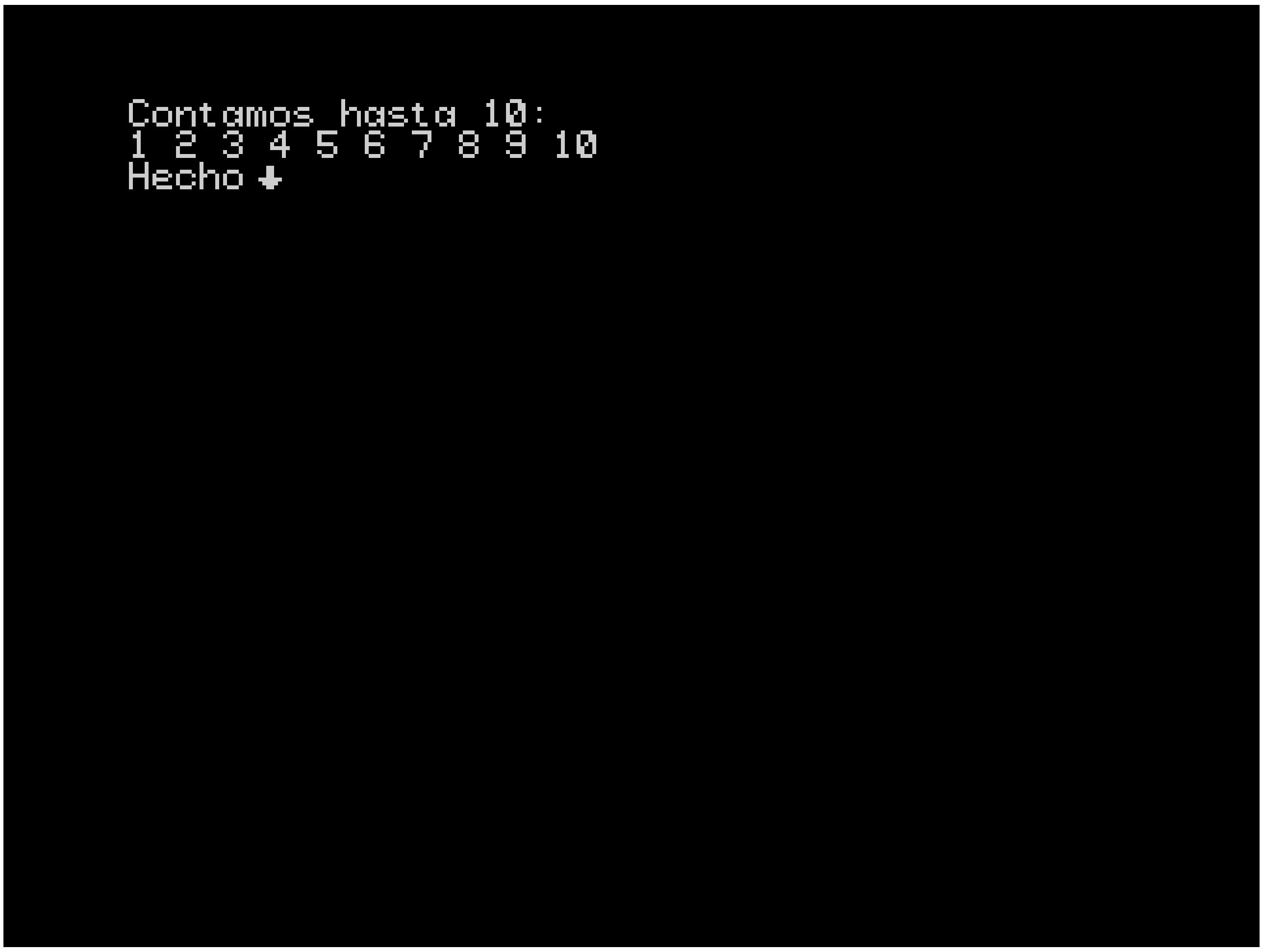
Loops

To make certain parts of the code repeat while a condition is met we have `WHILE (cond) ... WEND`:

```
[[ /* Pone colores de pantalla y la borra */
   PAPER 0    /* Color de fondo negro  */
   BORDER 0   /* Borde de color negro  */
   INK   7    /* Color de texto blanco */
   PAGEPAUSE 1
]]Contamos hasta 10:
[[
   DECLARE 0 AS cnt
   SET cnt TO 1
   WHILE (@cnt < 11)
      PRINT @cnt
      TAB 1
      SET cnt TO @cnt + 1
   WEND
]]
Hecho[[WAITKEY]]
```

In this example, I set the variable *cnt* to 1, and when the loop starts, it checks the condition that the variable is less than 11, as long as it is true, everything up to `WEND` is executed. Each time the loop is executed, we print the value of the variable, put a space with `TAB 1` and increment its value by 1. When it reaches 11, it stops executing and continues with whatever is after `WEND`.

And now it counts from one to 10:



```
Contamos hasta 10:
1 2 3 4 5 6 7 8 9 10
Hecho
```

It is important to define the entry and exit conditions well, and to keep in mind that the verification of the loop condition is always done **at the beginning** of each iteration.

With the addition of the loop, we can implement almost anything with the CYD language.

Text compression and abbreviations

To save disk space, the compiler compresses the text, looking for the most common substrings in the text and replacing them with tokens or abbreviations.

The `make_adv.cmd` script will make the compiler look for abbreviations if it does not find a file named `tokens.json` in its folder, and will save the abbreviations found in a file with that name. On the other hand, if it finds such a file, it will pass it as a parameter to the compiler so that it uses the abbreviations contained in it. To make it look for abbreviations again, simply delete the `tokens.json` file before running it.

The process of looking for abbreviations can be very expensive as the amount of text in the adventure increases. My advice is to write the adventure script *before* you start programming and put it all in a text file that you can feed to the compiler, so that it can generate a suitable abbreviation file. Once you have that, you can start adding commands to the text to shape the adventure. Once you have it *almost* finished, you can generate abbreviations again to see if you can carve out some more space.

Workflow

From this point on, you should have enough knowledge to be able to tackle your own adventure. From here on, I cannot give you advice on how to write your adventure, that is up to your imagination. However, I am going to recommend a workflow based on the experience of developing the adventure *The Rings of Saturn*, which I submitted to the Radastán 2023 Adventure Contest.

My first recommendation is to write **before** the bulk of your adventure, as you will have already read in the previous chapter. With that, you will have a valid list of abbreviations to start working on, since the process of searching for abbreviations is **very expensive**. To give you an idea, compressing the entire *The Rings of Saturn* (a book of about 150 pages) takes up to three quarters of an hour. In addition, our workflow is essentially going to consist of writing, compiling and testing over and over again, so we are interested in making this process as fast as possible.

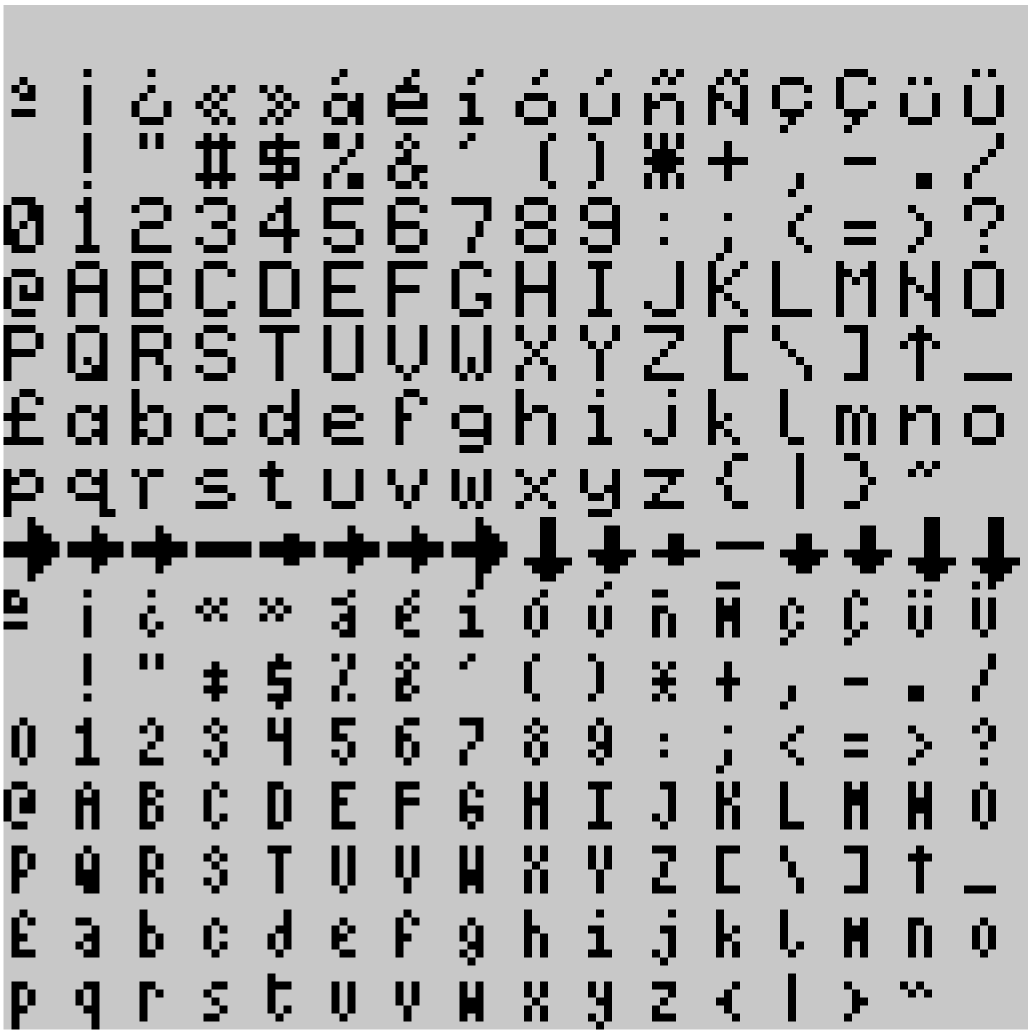
You can write your adventure with your favorite text editor, but always **in plain text**, since that is what the compiler understands.

An important issue when working with the editor is the encoding of the texts. A file on the disk is a collection of bits ordered with a name, the computer does not "know" what to do with those bits; it is the programs that interpret those bits.

A text file is a file whose bits represent characters. The way of interpreting those characters is called **encoding**. The most famous encoding used since the beginning of computing is **ASCII**, whose standard supports 128 characters, designed for the English language and with extensions to 256 for international and special characters. Today, ASCII has become too small, and the encoding used as standard by all current text editors by default is **UTF-8**.

All current editors support multiple encodings, so you will have to investigate the operation of your editor to select the correct encoding.

The **CYD** compiler supports UTF-8 text, but you have to take into account a number of limitations when writing your text or copying it from another source. The default character set of the engine is the following:



Those are the characters you have available, and UTF-8 has *millions* of different spellings, and I'm not exaggerating, so most of them will fail the compiler. In the manual I detail the only characters that will be translated from UTF-8 to the encoding used by the engine:

Character	Position
'ª'	16
'¡'	17
'¿'	18
'«'	19
'»'	20
'á'	21
'é'	22
'í'	23
'ó'	24
'ú'	25
'ñ'	26

Character	Position
'Ñ'	27
'ü'	28
'Ü'	29

As you can see, for example, the accented uppercase vowels are not there. The compiler will automatically transform the accented uppercase letters into unaccented uppercase letters.

Once you have written a good part of your adventure, and obtained an abbreviation file, we can start programming it.

My recommendation is to use the "divide and conquer" tactic, a technique that consists of dividing a problem to be solved into smaller subproblems that we will solve little by little. In our context, it means that we should divide the adventure into sections that we will program one by one. You have probably already instinctively done this step when writing your story, dividing it into chapters.

With this subdivision, we will now work with two source files, a *global file* with the text of the "complete" adventure, and another that we will pass to the compiler with the section that we are going to program, which we will call *working file*. The process consists of the following steps:

1. Copy one of the uncompleted sections from the global file to the working file.
2. Add the commands to the text of the working file and format it if necessary.
3. Compile the working file.
4. Run the resulting image in an emulator and test.
5. If you are not satisfied, modify the file and return to step 3.
6. When you have the complete section, copy the completed section in the working file and replace it in the global file.
7. If you still have sections to complete, return to step 1.
8. Move the entire global file to the working file.

However, once you have completed the adventure, you will probably have to fix things, compile and test again. And, depending on the length of the adventure, getting to the relevant part can be a hell. I will suggest several techniques to avoid this.

One tactic we can employ is to label the beginning of each and every section with `LABEL` (we would do this in the previous process). Then, we simply put a `GOTO` to the label of the section we want to test at the beginning of the adventure. When the section is executed, it would jump directly to the relevant section, saving us precious time. Remember to remove the initial `GOTO` afterwards!

Another technique I have employed is to cancel pauses. To do this, I use the compiler's ability to use comments within code sections. Using our text editor, we can replace all occurrences of the `WAITKEY` command, for example, with the text `/*WAITKEY*/`. When we compile again, since those commands are commented out, they will not be executed and everything will be displayed non-stop. When we want to undo the change, we do the opposite process, replacing `/*WAITKEY*/` with `WAITKEY`. To speed up this process even further, we can even introduce the commands commented out in the previous phase.

Finally, almost all modern emulators offer the possibility of speeding up the execution. We can take advantage of this to display the text faster than usual.

I hope these techniques help you create your adventure in the most comfortable way possible. Programming is an iterative process of writing, compiling, running and testing, and it can be monotonous, but also very satisfying when we get the desired result.

Side-scrolling menus

Option menus also allow *side-scrolling* and the `MENUCONFIG` command allows you to configure this behavior. First, it should be noted that everything explained earlier in the tutorial is still applicable, but it has become a special case that we will detail. Without going into too many technical details of the engine, we will explain how the menu system works now.

The menu system is based on a list of options. Each time we declare an option, the screen position (adjusted to 8x8 coordinates) and the corresponding jump address are saved in the list, so that the options are saved in the same order in which we declared them. When the screen is cleared or an option is chosen from the menu, the list is cleared.

When you activate the menu with `CHOOSE`, a pointer corresponding to the selected option is placed in the first position of the list, and the corresponding option in the list shows the selection icon in the stored screen position. When you scroll through the menu with the keys, you move that pointer along the list, and at each step, the on-screen pointer moves to the corresponding position.

In previous versions, you were only allowed to move one position forward in the list with the *A* key and one position backward with the *Q* key. Because of this key layout, you were expected to have vertical menus.

Now you are allowed to use the *O* and *P* keys to scroll *"horizontally". In fact **CYD** has no concept of horizontality and verticality, it simply scrolls through the list in a certain way according to the keys pressed. Thanks to the command `MENUCONFIG x,y` you can configure this behavior. The command admits two parameters that mean the following:

- The first parameter (which we will call `X`) determines the increment or decrement of the selected option number when you press **P** and **O** respectively.
- The second parameter (which we will call `Y`) determines the increment or decrement of the selected option number when we press **A** and **Q** respectively.

That is, if the selected option number at a given time is `N`, then:

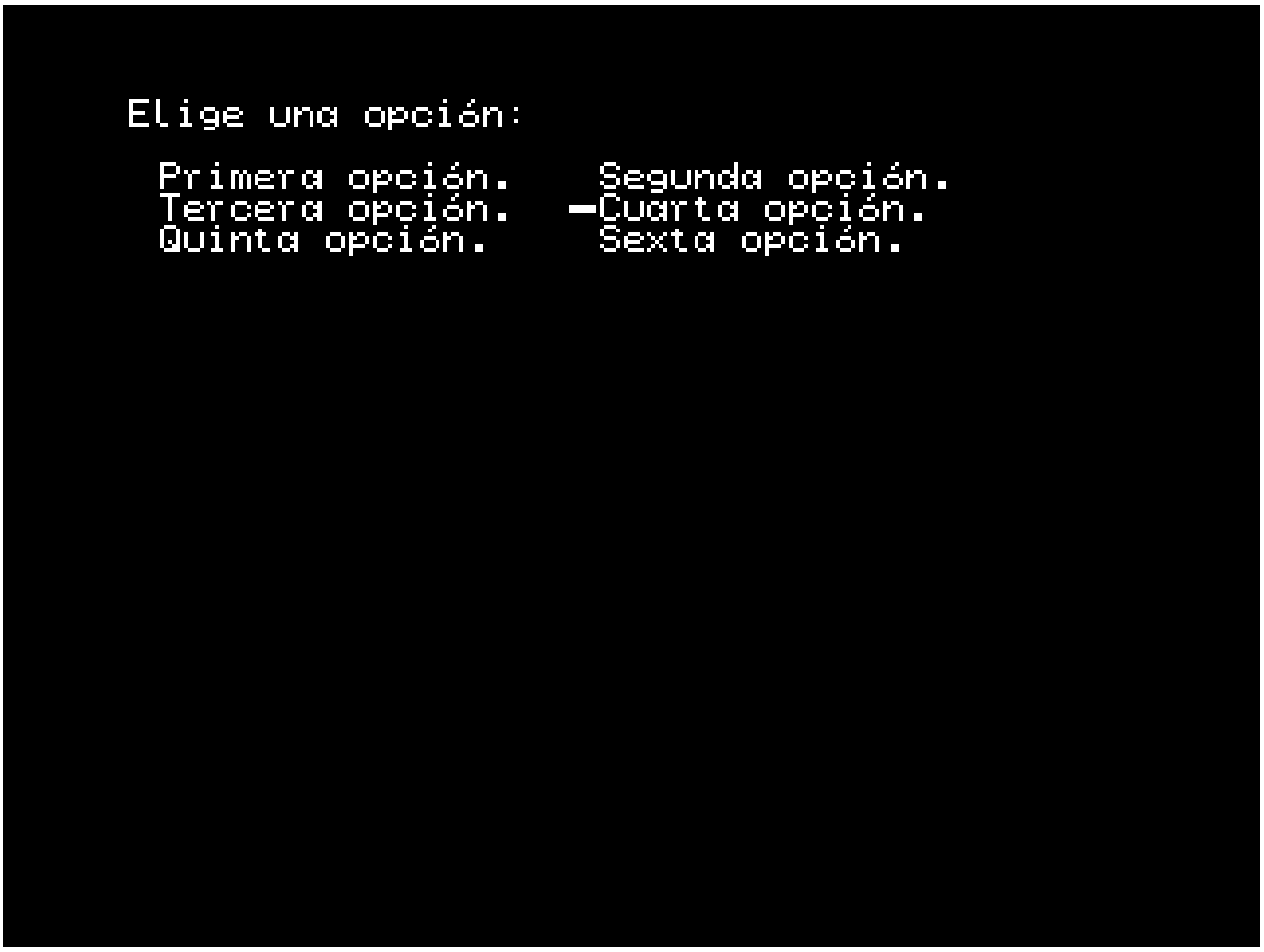
- When we press **O**, `N` becomes `N - X`.
- When we press **P**, `N` becomes `N + X`.
- When we press **Q**, `N` becomes `N - Y`.
- When we press **A**, `N` becomes `N + Y`.

With this we can make different types of movements, but it is our responsibility to place the options in the correct order and position on the screen so that it is consistent with the keystrokes.

Let's see it, as always, with the following example:

```
[[
    PAGEPAUSE 1
    BORDER 0
    INK 7
    BRIGHT 1
    FLASH 0
    PAPER 0
    CLEAR
]]Elige una opción:

[[OPTION GOTO Opcion1]]Primera opción.  [[OPTION GOTO Opcion2]]Segunda opción.
[[OPTION GOTO Opcion3]]Tercera opción.  [[OPTION GOTO Opcion4]]Cuarta opción.
[[OPTION GOTO Opcion5]]Quinta opción.   [[OPTION GOTO Opcion6]]Sexta opción.

[[
    MENUCONFIG 1,2 : CHOOSE
    LABEL Opcion1]]Has elegido la opción 1.[[
    GOTO Siguiente
    LABEL Opcion2]]Has elegido la opción 2.[[
    GOTO Siguiente
    LABEL Opcion3]]Has elegido la opción 3.[[
    GOTO Siguiente
    LABEL Opcion4]]Has elegido la opción 4.[[
    GOTO Siguiente
    LABEL Opcion5]]Has elegido la opción 5.[[
    GOTO Siguiente
    LABEL Opcion6]]Has elegido la opción 6.[[
    LABEL Siguiente
    WAITKEY : END]]
```

According to what we have explained, when using `MENUCONFIG 1,2`, then when we press **O** and **P**, the selected option is decremented and incremented by 1, and when we press **Q** and **A**, the selected option is decremented and incremented by 2. That is asking us for a menu with two elements per row, since when we press *down* or *up*, we jump two by two. With this, we must take care of placing the options in two columns and in order from left to right.



```
Elige una opción:

  Primera opción.    Segunda opción.
  Tercera opción.   ━Cuarta opción.
  Quinta opción.     Sexta opción.
```

The default behavior, which simulates the *vertical* behavior of previous versions, is done with `MENUCONFIG 0,1`. That is, when we press **O** and **P**, the selected option is decremented and incremented by 0, that is, **it does nothing**, it does not move. And when you press **Q** and **A**, the selected option is decremented and incremented by 1, so we have a strictly vertical menu.

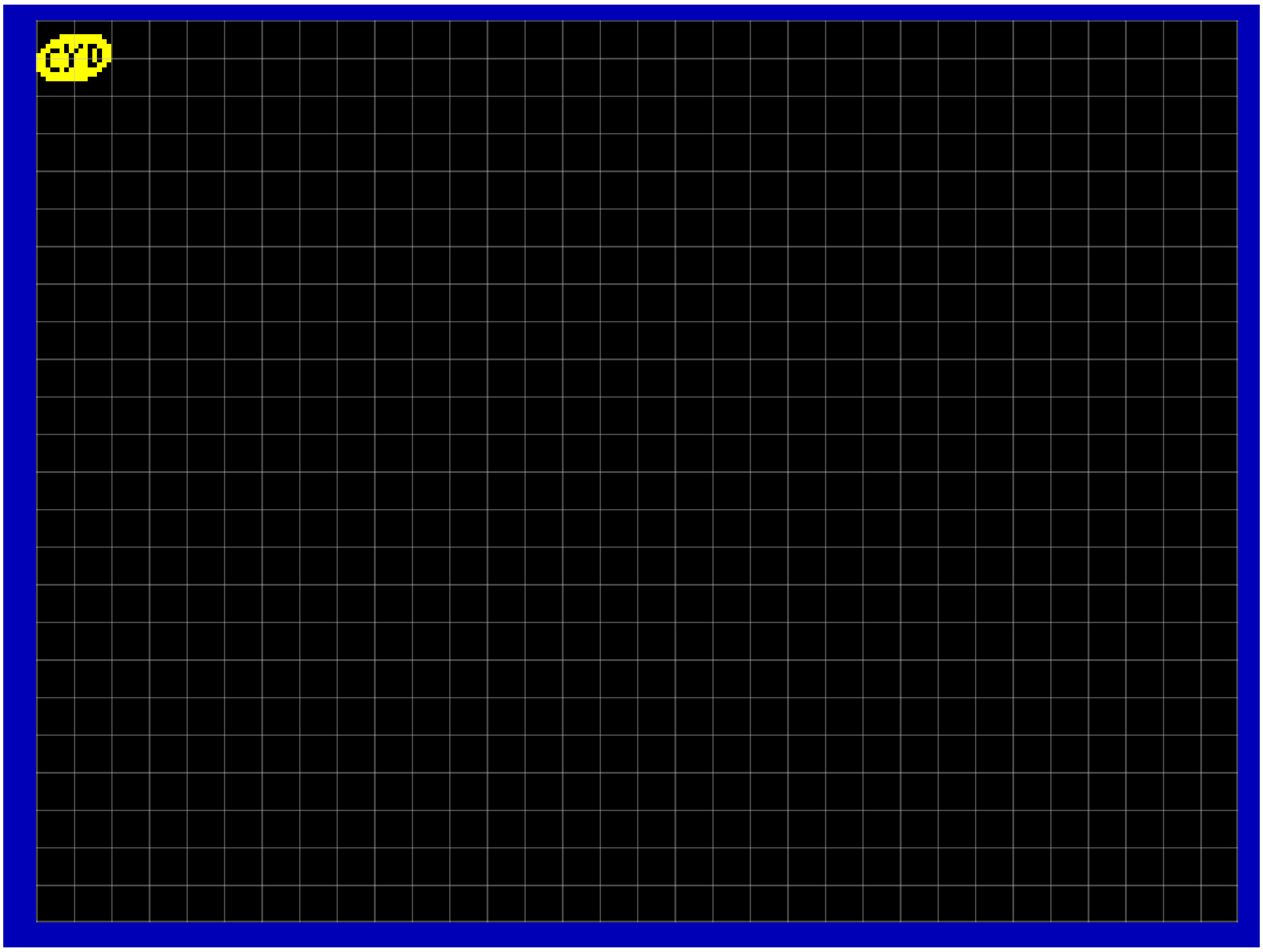
Thus, if with `MENUCONFIG 0,1` we will have a vertical menu in one column, with `MENUCONFIG 1,0` we will have a horizontal menu in one row. With `MENUCONFIG 1,3` we will have a three-column menu, etc. But remember that `MENUCONFIG` does not indicate the arrangement of the options, we define that when we declare them, but the behavior of the buttons.

In this case, it is better to try the example live and experiment on your own. Keep in mind that the pointer will never exceed the limits below zero, nor above the number of options in the list. Therefore, you should enter *reasonable* values or the menu will not work well.

Copying image fragments to screen

If you have followed the tutorial in order, you will already know that with the command `PICTURE` we load an image into the screen buffer, and with `DISPLAY` we display it, that is, it is copied from the buffer to the screen, but always in full screen. But it is also possible to copy a piece of the image loaded in the buffer to a certain defined position on the screen using the command `BLIT`.

But first, let's create the following SCR image with some graphical editor for Spectrum:



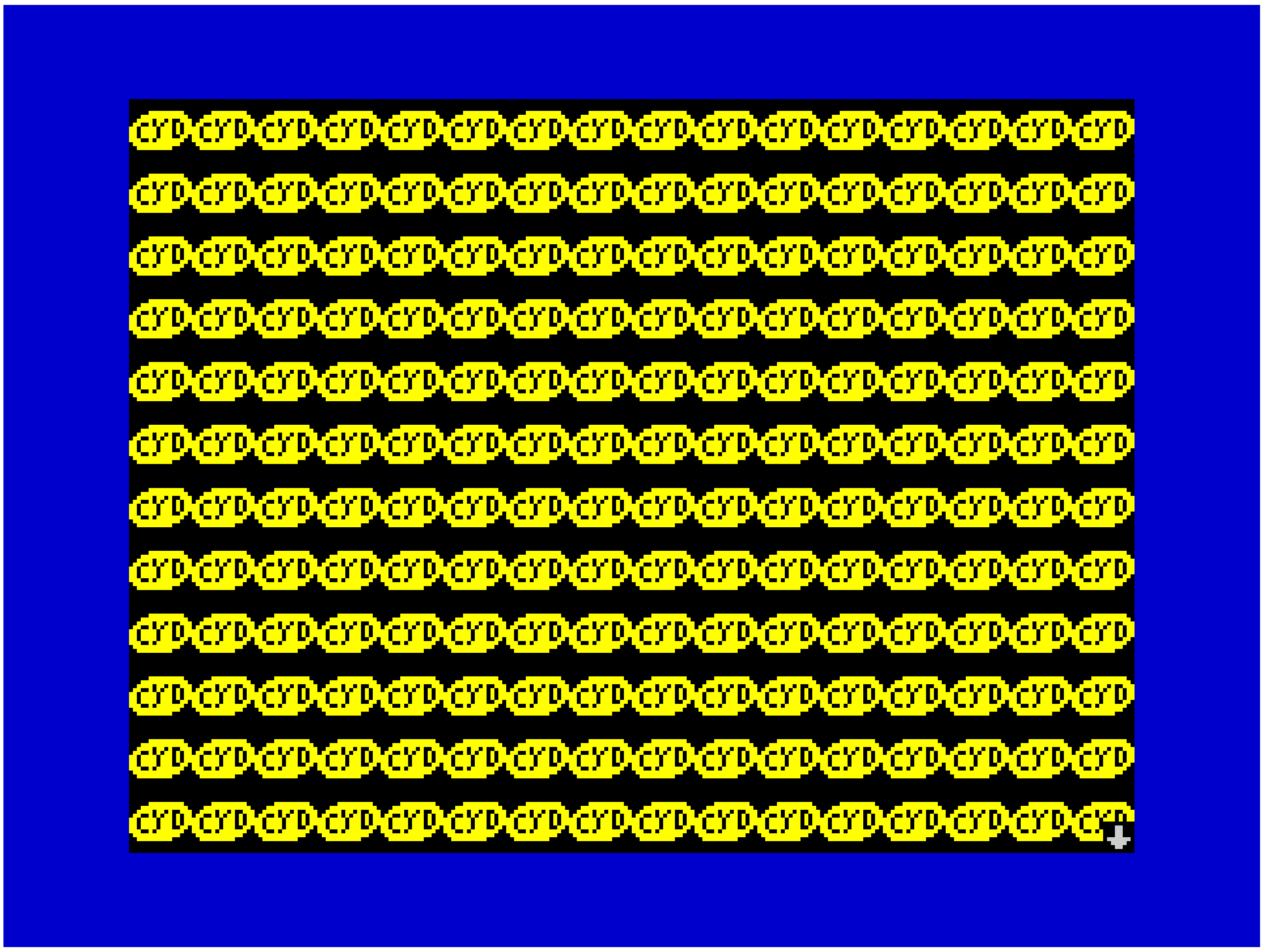
And we leave it in the directory `IMAGES`. You can see that we have drawn in the first block of 16x16 pixels, or what is the same, 2x2 characters.

Now, let's use the following example:

```
[[
    INK 7
    PAPER 0
    BORDER 1
    CLEAR
    PICTURE 0            /* Cargando imagen 0 en el buffer */
    DECLARE 0 AS row     /* Variable para filas */
    DECLARE 1 AS col     /* Variable para columnas */
    SET row TO 0
    WHILE (@row < 24)
        SET col TO 0
        WHILE (@col < 32)
            BLIT 0, 0, 2, 2 AT @col, @row /* Copiando a pantalla */
            SET col TO @col+2
        WEND
        SET row TO @row+2
    WEND
    AT 31, 23
    WAITKEY
    END]]
```

This example basically clears the screen and loads image 0 into the screen buffer (the one we created earlier), and then we go through the rows from 0 to 24 and the columns from 0 to 32, jumping two characters at a time. If you've followed this tutorial, you won't have any trouble understanding how it works.

The key component is the command `BLIT 0, 0, 2, 2 AT @col, @row`. What we're telling the engine is "take the rectangle from the buffer with origin (0,0) and size 2x2 characters and copy it to the screen at the position defined by the variables col and row". Since we are traversing the entire screen using two nested loops, the result is the following:



In general, with `BLIT x_orig, y_orig, width, height AT x_dest, y_dest`, what we do is copy from the image loaded in the buffer a piece of it defined by the first 4 parameters, and we copy it to the screen from the last two. This gives us many possibilities to build the image on the screen using "tiles", create animations, curtains, etc.

As final notes, three things to highlight:

- Variables can only be used in the last two parameters, as is the case in this example. The parameters that define the rectangle to be cut are always fixed.
- In all parameters, the unit is the 8x8 pixel character.
- The implementation is slow due to its genericity, so if you are tempted to use them as "sprites", the results will be disappointing due to flickering.

Reading the keyboard, variable arrays and indirections

First, I'll warn you that this is a fairly advanced chapter that introduces some rather advanced programming concepts if you've never programmed before. If you don't understand it, you can skip it without any problems.

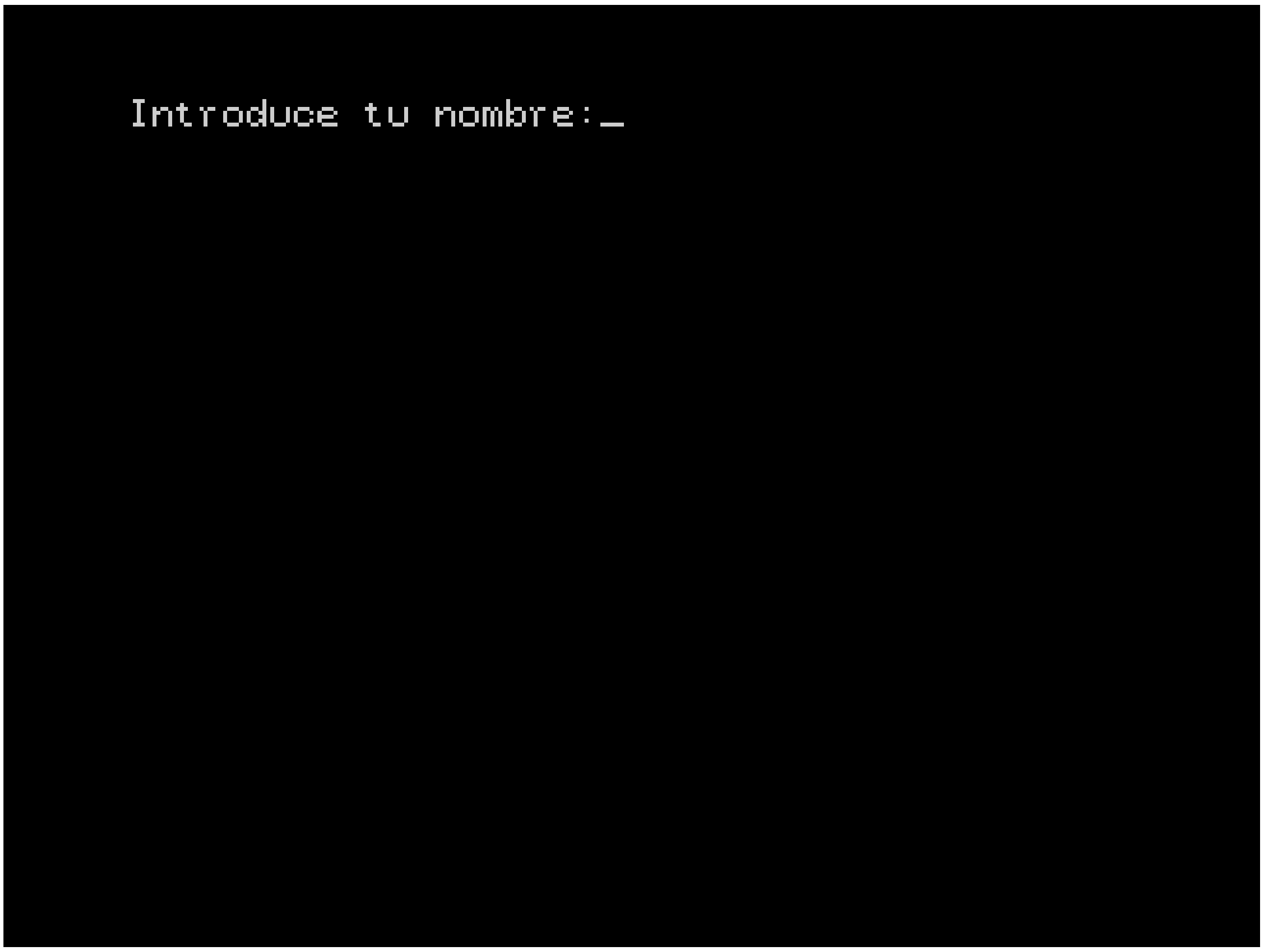
There are a series of commands that allow reading characters from the keyboard and deleting them on the screen, and, together with indirection, they allow us to store text strings in variables and use them as arrays. However, **BE CAREFUL**, we only have 256 variables available... This means that we can't store very long text strings, but we can use it to store a short one, like the name of our protagonist or to create a simple command line.

Let's see it with the following example:

```
[[ /* Pone colores de pantalla y la borra */
   PAPER 0    /* Color de fondo negro  */
   INK   7    /* Color de texto blanco */
   BORDER 0   /* Borde de color negro  */
   CLEAR      /* Borramos la pantalla*/
   PAGEPAUSE 1

   DECLARE 0 AS str        /* Comienzo del array de 16 caracteres */
   DECLARE 16 AS ptr       /* Puntero actual sobre la cadena */
   DECLARE 17 AS chr       /* Carácter Leído del teclado */

]]Introduce tu nombre:[[
   GOSUB inputStr]]
Bienvenido [[
   GOSUB printStr
   NEWLINE /* Salto de línea */
   WAITKEY
   END

   /* Subrutina para imprimir una cadena de 16 caracteres*/
   #printStr
   /* Inicializamos el puntero con la dirección de la variable inicial de la
cadena */
   SET ptr TO @@str
   /*
   Mientras el puntero sea menor a la dirección del mismo...
   (La dirección nos sirve como marcador para indicar el final de la cadena)
   */
   WHILE (@ptr < @@ptr)
      /* Si el contenido de la variable marcada con el puntero es cero acabamos */
      IF [@ptr] = 0 THEN RETURN ENDIF
      /* Imprimimos el carácter haciendo indirección sobre el puntero */
      CHAR [@ptr]
      /* Incrementamos el puntero una posición */
      SET ptr TO @ptr + 1
   WEND
   RETURN

   #inputStr
   /* Inicializamos el puntero con la dirección de la variable inicial de la
cadena */
```

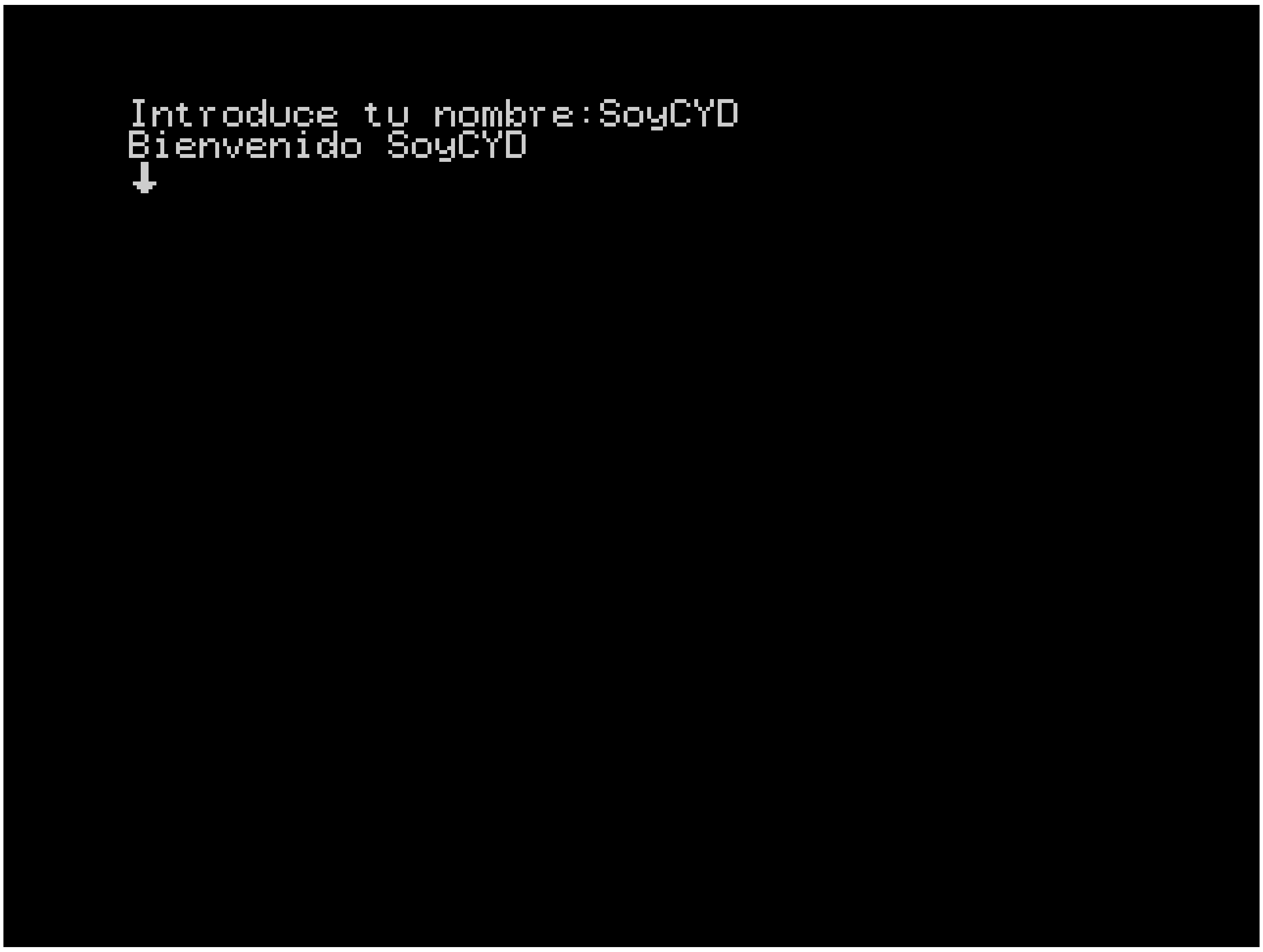
```
   SET ptr TO @@str
   /* Rellenamos toda la cadena con ceros */
   WHILE (@ptr < @@ptr)
      SET [ptr] TO 0
      SET ptr TO @ptr + 1
   WEND
   /* Volvemos a poner el puntero al principio */
   SET ptr TO @@str
   /* Bucle infinito */
   WHILE ()
      /* Ponemos el carácter '_' como cursor */
      CHAR 95
      /* Leeemos una tecla pulsada y guardamos su código ASCII en chr */
      SET chr TO INKEY()
      /* Borramos el cursor */
      BACKSPACE
      /* Si la tecla es ENTER, salimos */
      IF @chr = 13 THEN RETURN ENDIF
      /* Si la tecla es DELETE... */
      IF @chr = 12 THEN
         /* Ponemos el puntero actual a cero, si no estamos al final del array */
         IF @ptr < @@ptr THEN
            SET [ptr] TO 0
         ENDIF
         /* Si no estamos al principio del array... */
         IF @ptr > @@str THEN
            /* Borramos la posición actual en pantalla y volvemos atrás */
            BACKSPACE
            /* Movemos el puntero a la posición anterior */
            SET ptr TO @ptr - 1
            /* Lo rellenamos también con cero */
            SET [ptr] TO 0
         ENDIF
      ELSE
         /*
         Si el carácter es mayor que 32 y menor que 128 (caractéres ASCII
imprimibles)
         y no estamos al final del array...
         */
         IF @chr >= 32 AND @chr < 128 AND @ptr < @@ptr THEN
            /* Imprimimos el carácter */
            CHAR @chr
            /* Guardamos el carácter en el array */
            SET [ptr] TO @chr
            /* Avanzamos el puntero */
            SET ptr TO @ptr + 1
         ENDIF
      ENDIF
   WEND
]]
```

As always, if we try it:



```
Introduce tu nombre:_
```

A "command line" appears where we can type; with **ENTER** we validate, and with **DELETE** we can delete the last character we have typed. You will notice, if you play with the program, that if we exceed 16 characters, it does not let us enter more, and only allows us to delete the last character or press **ENTER**. If we do the latter:



```
Introduce tu nombre:SoyCYD
Bienvenido SoyCYD
```

It prints the string we have entered! In this simple way, we already have a couple of subroutines that we can use to read and print small text strings. The comments included are enough to know what it is doing at each step, but there are things that we need to consolidate and with concepts that are already quite complex; so this section will have, exceptionally, subsections.

Indirection

In the example code you will have seen that variables are placed in square brackets, for example `SET [ptr] TO 0`. This is called indirection, and what it means is that **the result of the expression inside the square brackets is used as the pointer to the variable to be accessed**. This concept is called **indirection**, which allows us to create **pointers** and implement **arrays**.

The difference between `SET 1 TO 0` and `SET [1] TO 0` is that the first command means "*store zero in variable 1*", and the second means "*store zero in the variable whose number corresponds to the content of variable 1*". So, if in variable 1 we had a two, then it will store zero in variable 2. In this case, we are using variable number 1 as a *pointer*, since variable 1 *points* to another variable.

Let's look at an example. We want to fill variables 0 through 7 with zeros, using variable number 8 as a pointer. The code to do this would be as follows:

```
[[
SET 8 TO 0
WHILE (@8 < 8)
   SET [8] TO 0
   SET 8 TO @8 + 1
WEND
]]
```

In the first line we initialize the pointer to 0 with `SET 8 TO 0`, which would give us this situation:

```
         0 1 2 3 4 5 6 7 8
        +-+-+-+-+-+-+-+-+-+-
        | | | | | | | | |0|...
        +-+-+-+-+-+-+-+-+-+-
```

Already inside the loop, we see that we have `SET [8] TO 0`, which indicates that in the variable whose index corresponds to the value of the content of variable 8, we will save the value zero:

```
         0 1 2 3 4 5 6 7 8
        +-+-+-+-+-+-+-+-+-+-
        |0| | | | | | | |0|...
        +-+-+-+-+-+-+-+-+-+-
         ^               |
         |               |
         +---------------+
```

The next step is to increment the pointer by one (`SET 8 TO @8 + 1`):

```
         0 1 2 3 4 5 6 7 8
        +-+-+-+-+-+-+-+-+-+-
        |0| | | | | | | |1|...
        +-+-+-+-+-+-+-+-+-+-
```

Since the value of variable 8 is still less than 8, we execute `SET [8] TO 0` again and save 0 in variable 1:

```
         0 1 2 3 4 5 6 7 8
        +-+-+-+-+-+-+-+-+-+-
        |0|0| | | | | | |1|...
        +-+-+-+-+-+-+-+-+-+-
           ^             |
           |             |
           +-------------+
```

We increase again:

```
         0 1 2 3 4 5 6 7 8
        +-+-+-+-+-+-+-+-+-+-
        |0|0| | | | | | |2|...
        +-+-+-+-+-+-+-+-+-+-
```

This process is executed until the loop condition is no longer met, which would result in this:

```
         0 1 2 3 4 5 6 7 8
        +-+-+-+-+-+-+-+-+-+-
        |0|0|0|0|0|0|0|0|8|...
        +-+-+-+-+-+-+-+-+-+-
```

So far I have used numerical names for variables. But how would we do it with variable name declarations?:

```
[[
DECLARE array AS 0
DECLARE ptr AS 8

SET ptr TO 0
WHILE (@ptr < 8)
   SET [ptr] TO 0
   SET ptr TO @ptr + 1
WEND
]]
```

The previous example could work... but if we decide to move the array and its pointer to other variables we will have a problem because in addition to the declarations, we would have to change `SET ptr TO 0` and `WHILE (@ptr < 8)` as well. But we have the following possibility:

```
[[
DECLARE array AS 0
DECLARE ptr AS 8

SET ptr TO @@array
WHILE (@ptr < @@ptr)
   SET [ptr] TO 0
   SET ptr TO @ptr + 1
WEND
]]
```

You will notice that instead, we are using the variable names with two @ signs `@@`. As you may know, when we find an @ sign inside an expression, it indicates that we are going to access the content of the corresponding variable. Well, the two @ signs indicate that we are going to use **the variable number** in the expression. So in the previous example `@@array` will be 0 and `@@ptr` will be 8. On the other hand, if we did the following, `@@array` would be 8 and `@@ptr` would be 16:

```
[[
DECLARE array AS 8
DECLARE ptr AS 16

SET ptr TO @@array
WHILE (@ptr < @@ptr)
   SET [ptr] TO 0
   SET ptr TO @ptr + 1
WEND
]]
```

It also allows us to access individual elements of the array. For example, to print the value of the fourth position of the array, such that if *array* is the variable zero:

```
       0 1 2 3 4 5 6 7
      +-+-+-+-+-+-+-+-+-
      |0|0| |4| | | | |...
      +-+-+-+-+-+-+-+-+-
             ^
```

We would do it like this with `PRINT [@@array + 3]`, and it would print 4. Remember that arrays are counted from zero, not from one. Another way to look at the `@@` operation is as the inverse of `[]`, so that `PRINT [@@array]` would be equivalent to `PRINT @array`.

This is a pretty advanced concept for someone who has never programmed before. I've tried to give a simple explanation, but if you don't understand it, it's normal and not necessary to make an interesting adventure, but if you do understand it, it can give you a powerful tool.

With these concepts in place and with the comments, you can understand how the first example works.

Reading the keyboard (INKEY)

To read a pressed key, we have the `INKEY()` function, which returns the code of the pressed key. This function works the same as its equivalent in Sinclair BASIC, but with the exception that the BASIC version returns zero if there is no valid key pressed, while the default behavior in CYD is to wait for a valid key to be pressed. If we want to reproduce the behavior of Sinclair BASIC, we can do so by using `INKEY(1)`.

The value returned when a key is pressed corresponds (normally) to its **ASCII** value, but adapted to the **Sinclair** version.

In the initial example, we check if the value of the returned key is 13 (ENTER) to validate and 12 (DELETE) to delete. If it is a value between 32 and 127, then it is a character that can be printed.

Deleting characters and making line breaks (BACKSPACE and NEWLINE)

To implement a command line, it is necessary to delete in case of a mistake. For this purpose, the `BACKSPACE` command has been added, which moves the cursor one position backwards and deletes the content of the new position. If it is on the left side of the defined border, it will jump one line backwards and will be placed at the end of the previous line (if possible).

It should be noted that the size used in the operation is that of character number 32 (the space). This is important if characters of sizes other than the space are used, the command will not delete them correctly, since CYD has no "memory" of what has already been printed.

The `NEWLINE` command is also introduced, which prints a line break without having to switch from "command" mode to "text" mode.

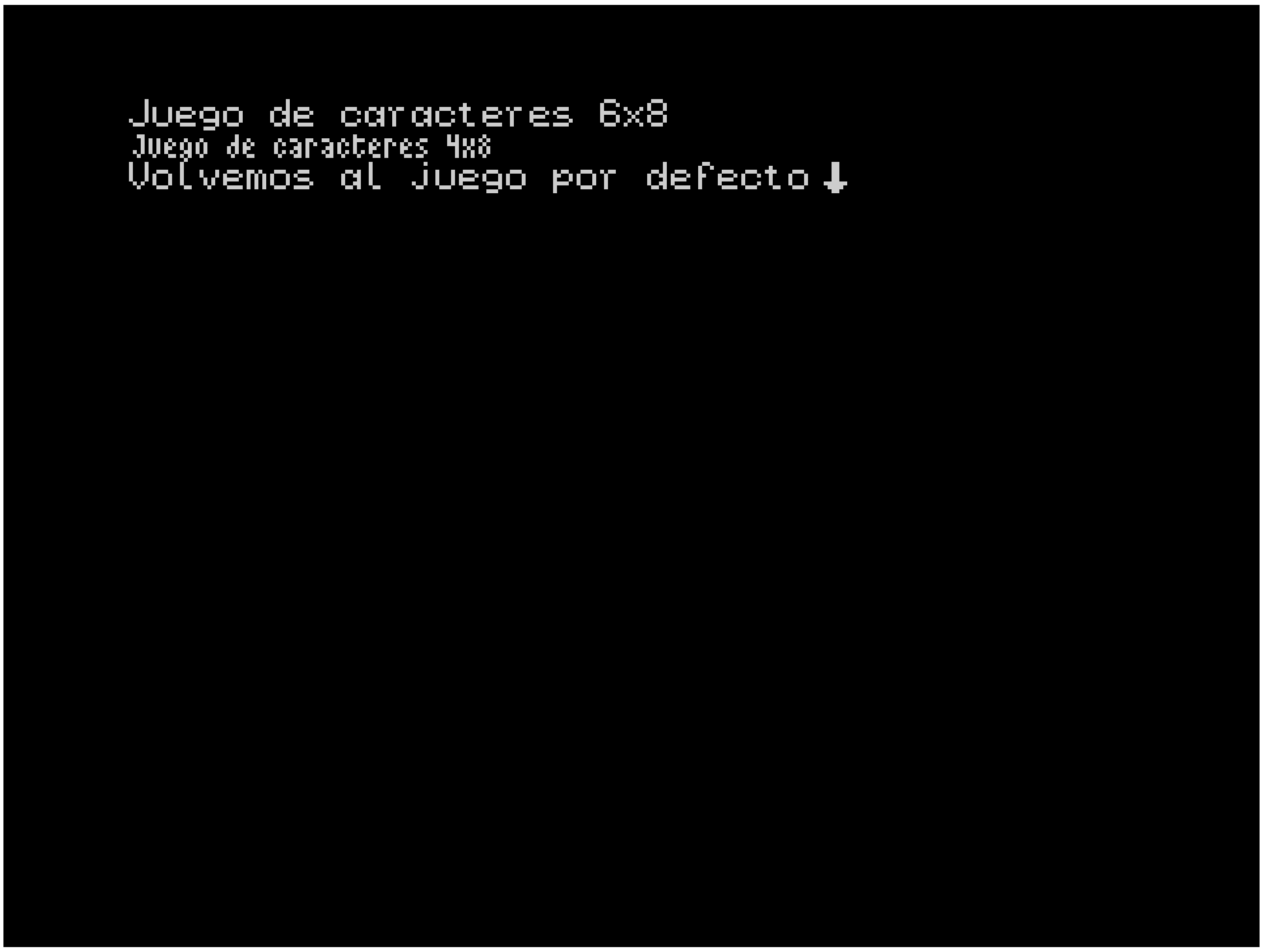
Both `NEWLINE` and `BACKSPACE` allow an optional parameter to indicate the number of times to print, such that `NEWLINE 3` would make 3 line breaks and `BACKSPACE 2` would delete two positions.

Using the alternative character set

CYD comes with two character sets for texts, the default one (which is 6 pixels wide) and an alternative one which is 4 pixels wide. To change from one to the other, use the `CHARSET` command as follows:

```
[[CHARSET 0]]Juego de caracteres 6x8
[[CHARSET 1]]Juego de caracteres 4x8
[[CHARSET 0]]Volvemos al juego por defecto[[
   WAITKEY
   END]]
```

With this result:


```
Juego de caracteres 6x8
Juego de caracteres 4x8
Volvemos al juego por defecto
```

The default character set (`CHARSET 0`) is characters from zero to 127, which contain characters 6 pixels wide. With `CHARSET 1`, we indicate to use characters from 128 to 255, which contain characters 4 pixels wide.

Windows

Windows are up to 8 different independent text areas that we can define on the screen. In each one we can define the margins differently, and each one has its own cursor position and attributes. At any given time, we can only have one "active" window, that is, one window to write on and configure. You switch between them using the `WINDOW` command. The default active window is window zero.

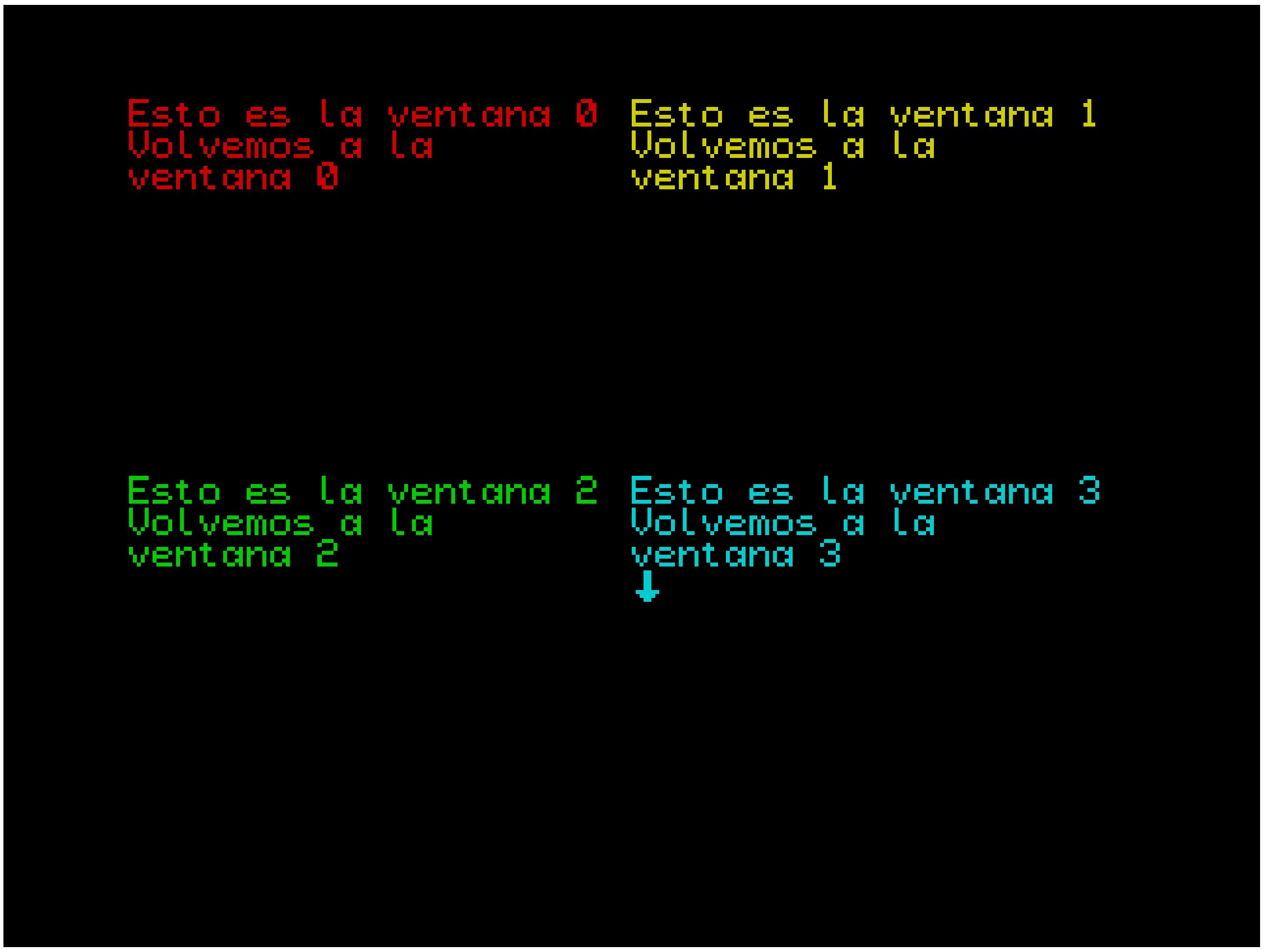
As always, an example is more eloquent:

```
[[/* La ventana por defecto es la cero */
  MARGINS 0, 0, 16, 12
  INK 2
  CLEAR
  WINDOW 1
  MARGINS 16, 0, 16, 12
  INK 6
  CLEAR
  WINDOW 2
  MARGINS 0, 12, 16, 12
  INK 4
  CLEAR
  WINDOW 3
```

```
    MARGINS 16, 12, 16, 12
    INK 5
    CLEAR
    WINDOW 0
  ]]Esto es la ventana 0
  [[
    WINDOW 1
  ]]Esto es la ventana 1
  [[
    WINDOW 2
  ]]Esto es la ventana 2
  [[
    WINDOW 3
  ]]Esto es la ventana 3
  [[
    WINDOW 0
  ]]Volvemos a la ventana 0
  [[
    WINDOW 1
  ]]Volvemos a la ventana 1
  [[
    WINDOW 2
  ]]Volvemos a la ventana 2
  [[
    WINDOW 3
  ]]Volvemos a la ventana 3
  [[
    WAITKEY
    END]]
```

In the example, 4 windows are used, from 0 to 3. First we start with window zero, which is selected by default, and we define its position, size and ink color. Then we switch to window 1 and do the same, and so on with the rest. Finally we switch between windows to put text in each one.

And this is the result:



```
Esto es la ventana 0   Esto es la ventana 1
Volvemos a la          Volvemos a la
ventana 0              ventana 1




Esto es la ventana 2   Esto es la ventana 3
Volvemos a la          Volvemos a la
ventana 2              ventana 3
```

As you can see, the color, margins, and cursor position are preserved in each window, and we can easily switch between them. This allows us to have different text areas for menus, descriptions, bookmarks, etc.

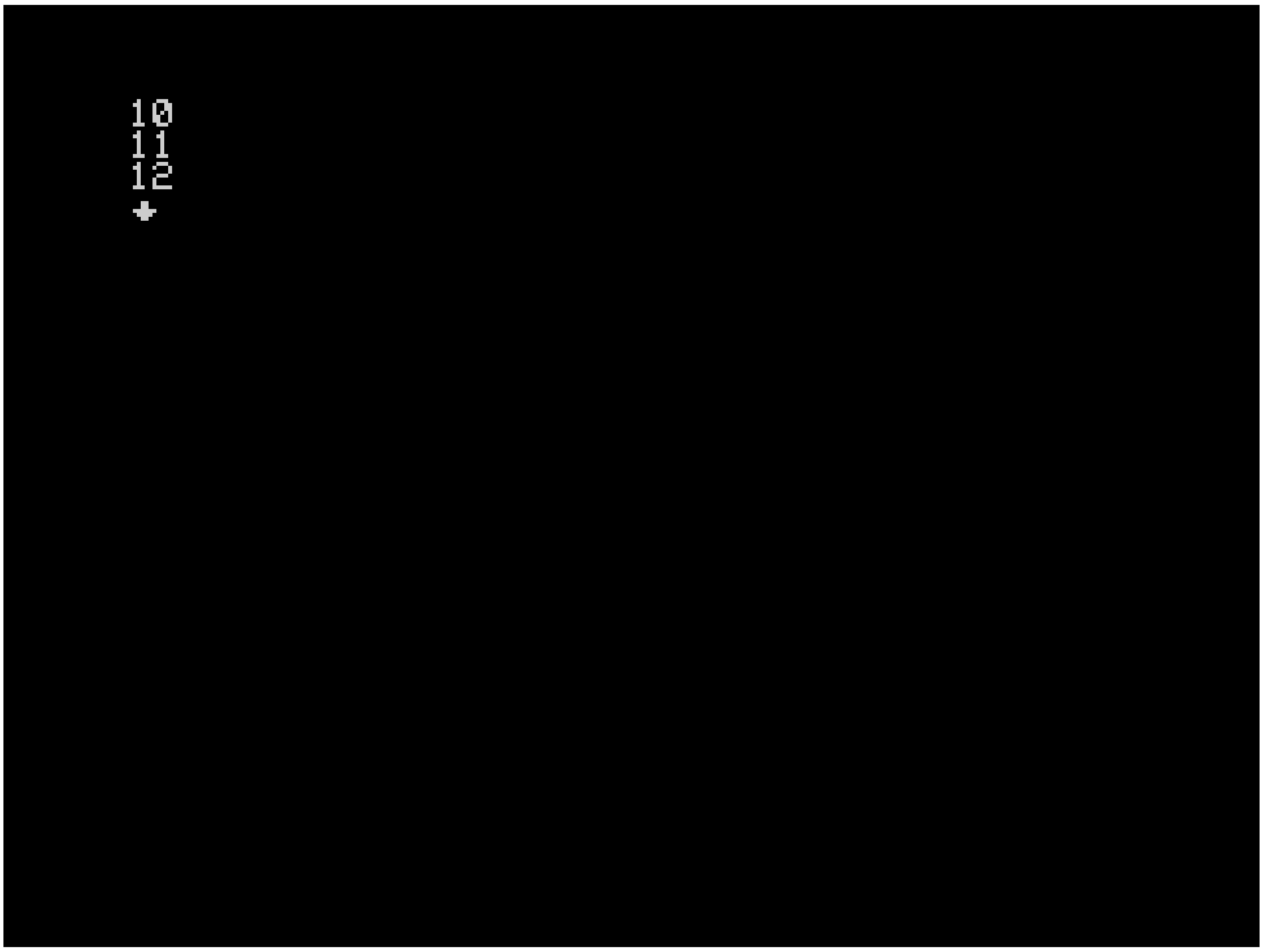
Finally, it should be noted that these windows are not equivalent to the windows of a desktop environment, such as Windows, Mac, KDE, or GNOME. If two windows overlap, and you type in one of them, the content of the other window is not preserved. Think of them as text areas instead.

Arrays or sequences

Arrays are sequences of numbers that can be accessed by an index. Using the `DIM` command we can declare an "array", so that with `DIM name(size)` we declare an array with the name `name` and size `size`. The size of an array cannot be greater than 256 or be zero. We access each of the elements of the array with the nomenclature `name(pos)`, where `pos` is the position number of the element to be accessed within the array, starting from zero.

Let's see an example:

```
[[
  DIM miArray(3)          /* Declaramos un array de 3 elementos del 0 al 2
*/
  LET miArray(0) = 10     /* Asignamos valores a cada elemento
*/
  LET miArray(1) = 11
  LET miArray(2) = 12
  PRINT miArray(0)        /* Imprimimos el valor almacenado en la posición 0
*/
  NEWLINE
  PRINT miArray(1)        /* Imprimimos el valor almacenado en la posición 1
*/
  NEWLINE
  PRINT miArray(2)        /* Imprimimos el valor almacenado en la posición 2
*/
  NEWLINE : WAITKEY]]
```



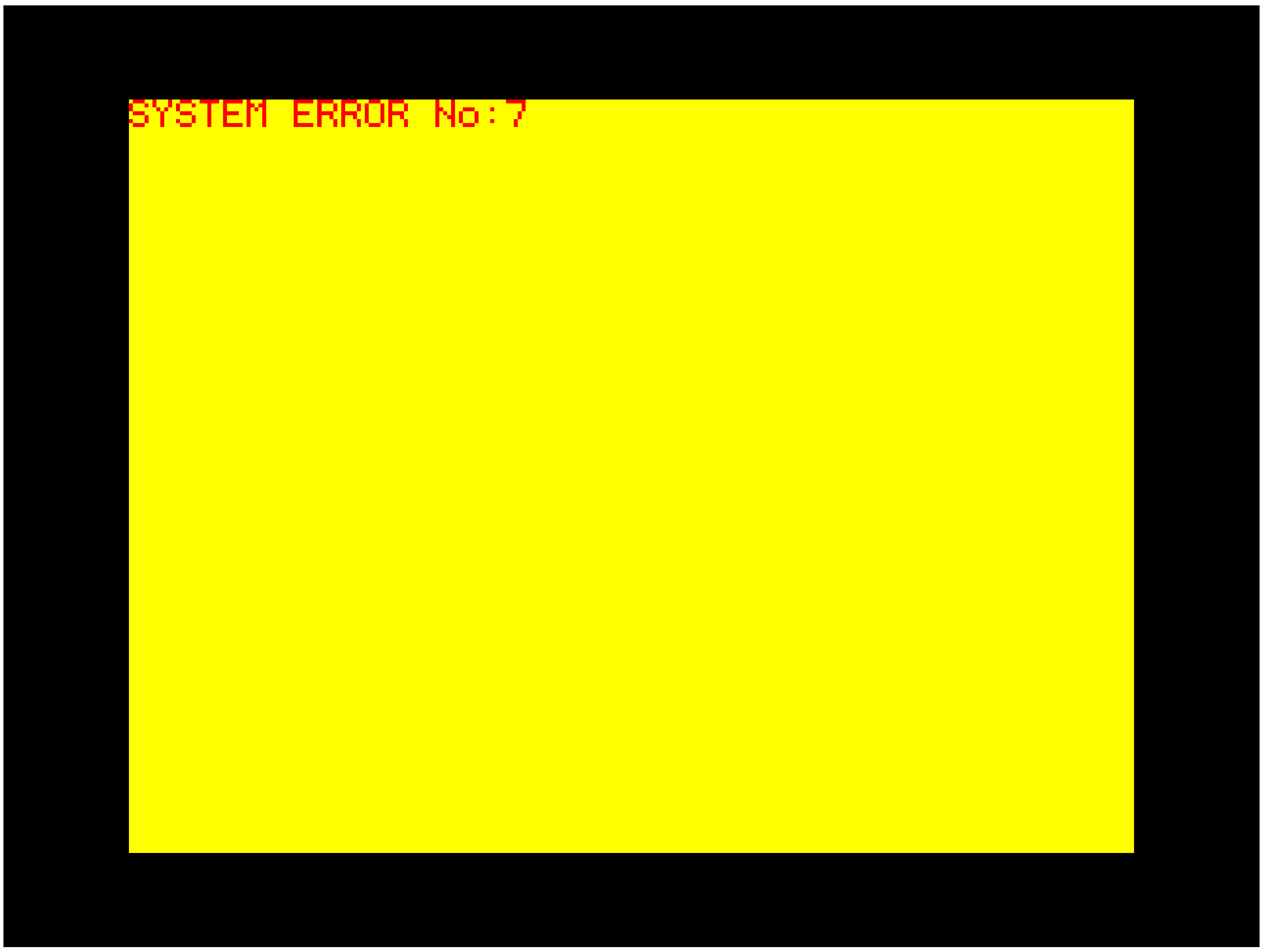
```
10
11
12
```

If you notice, we are treating `myArray(0)`, for example, as if it were a variable. It can be assigned and its values collected.

But **be careful!**, arrays start counting from zero to the size we have defined in the array declaration minus one. If we try to access a position outside its range, we will get a type 7 error:

```
[[
  DIM miArray(3)          /* Declaramos un array de 3 elementos del 0 al 2
*/
  LET miArray(0) = 10     /* Asignamos valores a cada elemento
*/
  LET miArray(1) = 11
  LET miArray(2) = 12
  PRINT miArray(0)        /* Imprimimos el valor almacenado en la posición 0
*/
```

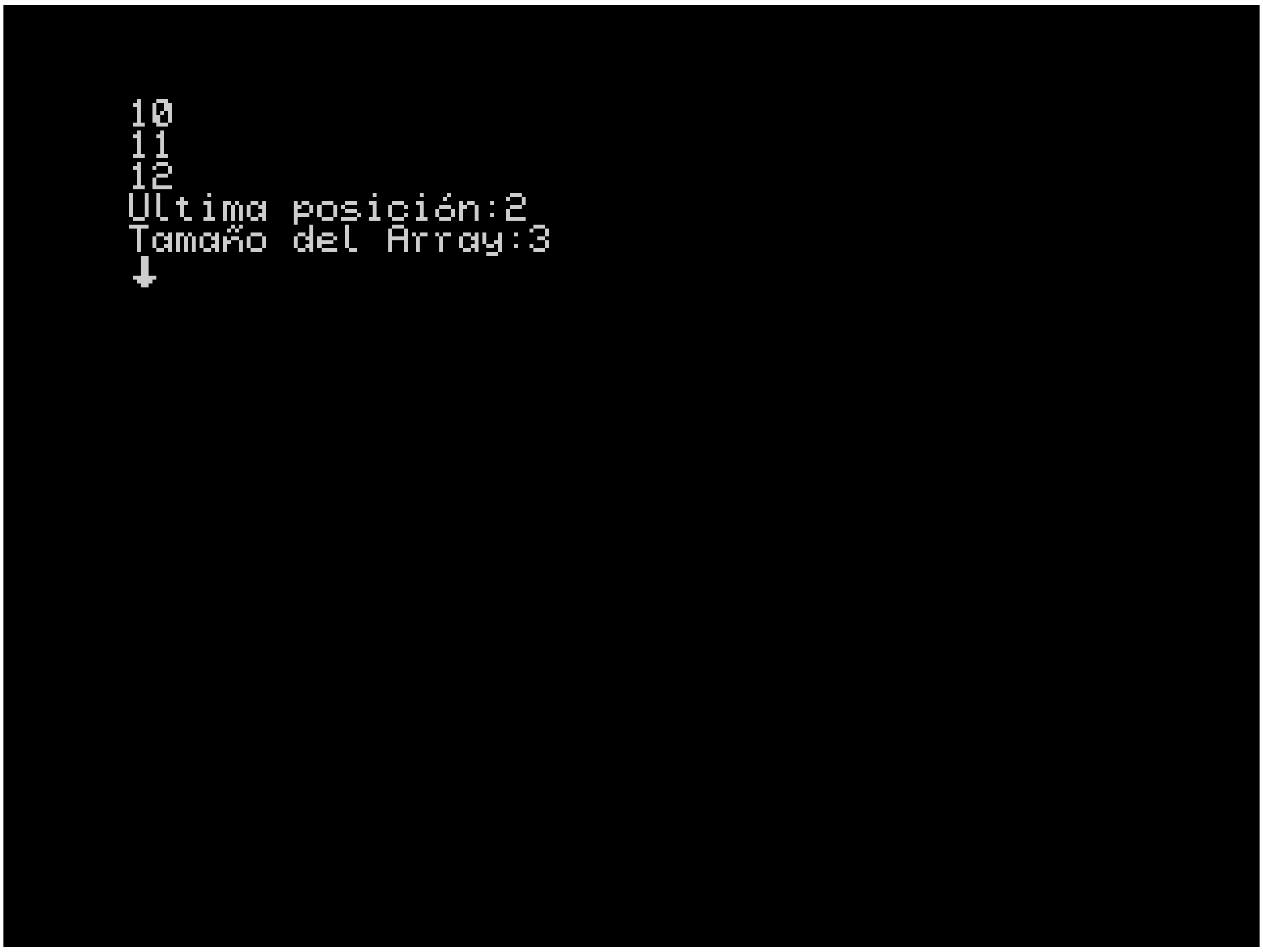
```
  NEWLINE
  PRINT miArray(1)        /* Imprimimos el valor almacenado en la posición 1
*/
  NEWLINE
  PRINT miArray(2)        /* Imprimimos el valor almacenado en la posición 2
*/
  NEWLINE
  PRINT miArray(3)        /* ¡ESTO DA ERROR!
*/
  NEWLINE : WAITKEY]]
```



```
SYSTEM ERROR No:7
```

To avoid these situations, you have the `LASTPOS(array_name)` function that returns the last allowed position for the given array so you can check before assigning:

```
[[
  DIM miArray(3)          /* Declaramos un array de 3 elementos del 0 al 2
*/
  LET miArray(0) = 10     /* Asignamos valores a cada elemento
*/
  LET miArray(1) = 11
  LET miArray(2) = 12
  PRINT miArray(0)        /* Imprimimos el valor almacenado en la posición 0
*/
  NEWLINE
  PRINT miArray(1)        /* Imprimimos el valor almacenado en la posición 1
*/
  NEWLINE
  PRINT miArray(2)        /* Imprimimos el valor almacenado en la posición 2
*/
  NEWLINE
  ]]Última posición:[[
  PRINT LASTPOS(miArray)  /* Vemos la última posición permitida     */
  ]]
  Tamaño del Array:[[
  PRINT LASTPOS(miArray)+1  /* Sumándo uno, tenemos su tamaño total */
  NEWLINE : WAITKEY]]
```



```
10
11
12
Ultima posición:2
Tamaño del Array:3
```

Arrays allow us to have tables of values, but initializing elements one by one can be cumbersome and inelegant.

We can initialize array values by declaring them in the following way:

```
[[ DIM precios(5) = {10, 40, 100, 200, 250} ]]
```

In fact, we can save ourselves from indicating the size since it will be calculated from the number of elements indicated:

```
[[ DIM precios() = {10, 40, 100, 200, 250} ]]
```

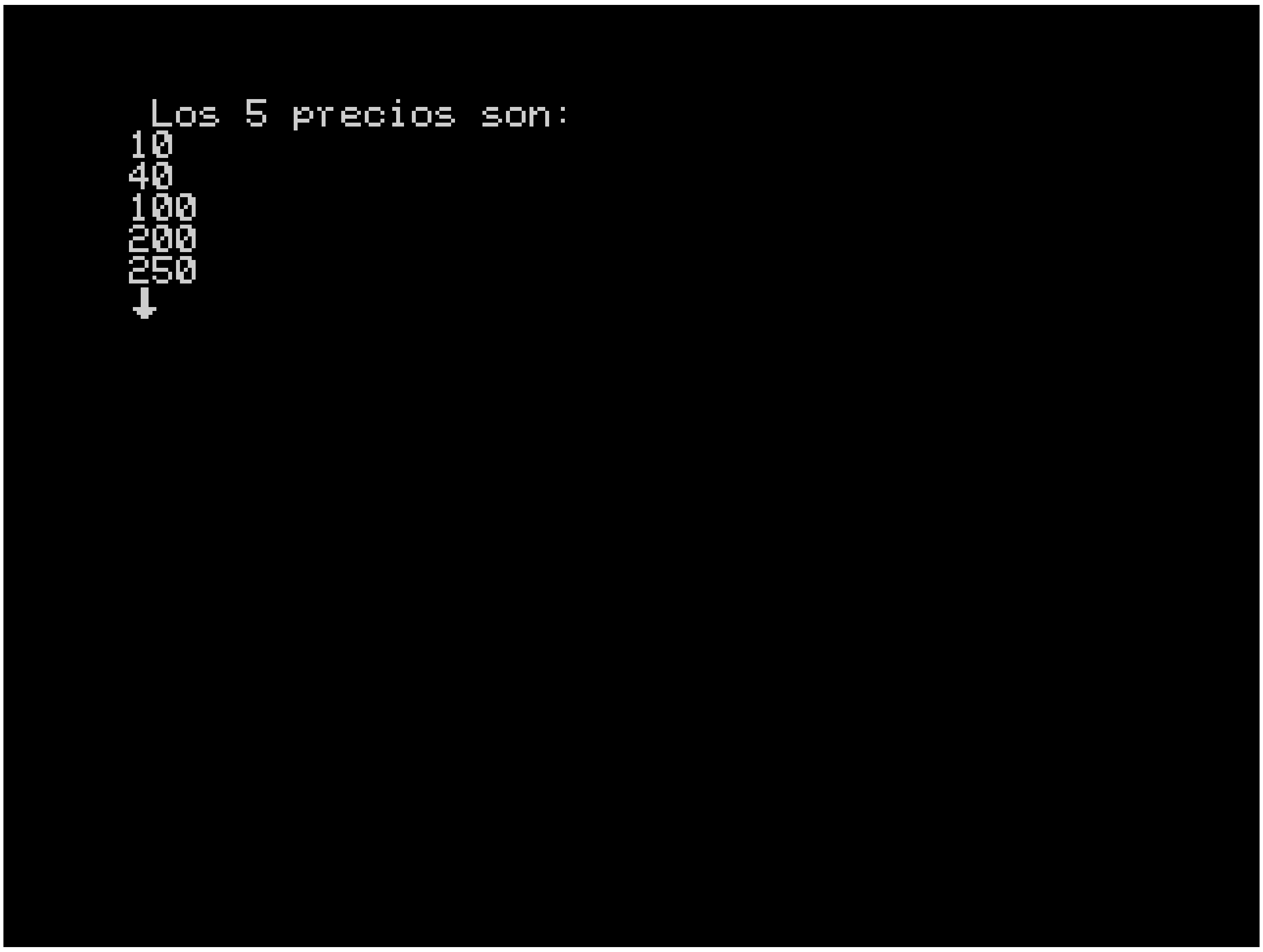
Note that in this case the array will have the size of the number of elements that we have indicated **and no more**.

Arrays cannot be resized and only accept values between 0 and 255, in the same way as variables. The maximum size is 256 and the minimum is 0.

Let's look at one last example to consolidate concepts:

```
[[
  DECLARE 0 AS c
  DIM precios(5) = {10, 40, 100, 200, 250}
]] Los [[ PRINT LASTPOS(precios)+1 ]] precios son:
[[
  LET c = 0
  WHILE(@c <= LASTPOS(precios))
      PRINT precios(@c)
      NEWLINE
      LET c = @c + 1
  WEND
  WAITKEY
]]
```

In the example, we use the variable `c` to iterate through the prices array until its last position and print the contents of each one:

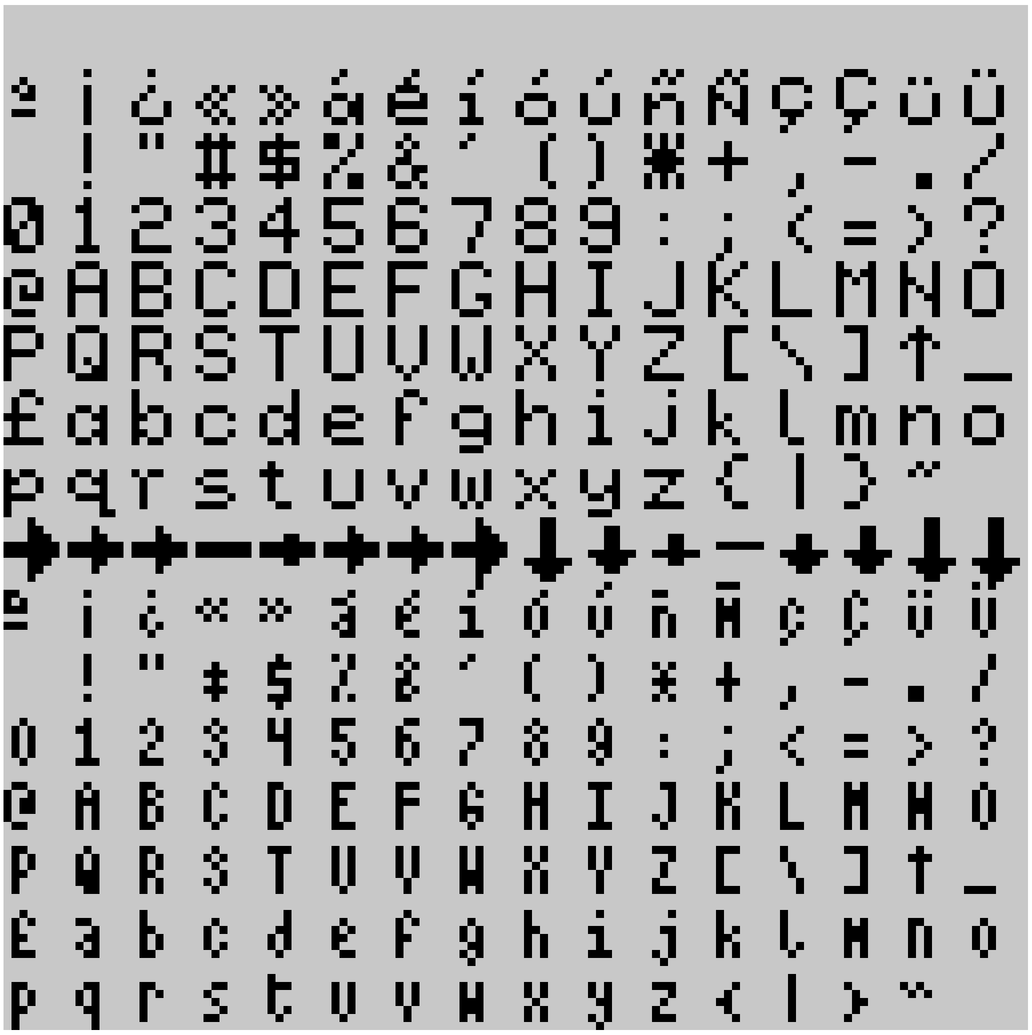


```
 Los 5 precios son:
10
40
100
200
250
```

Finally, it should be noted that another limitation that arrays have is that they cannot be recorded directly onto tape or disk, unless their contents are dumped into variables beforehand.

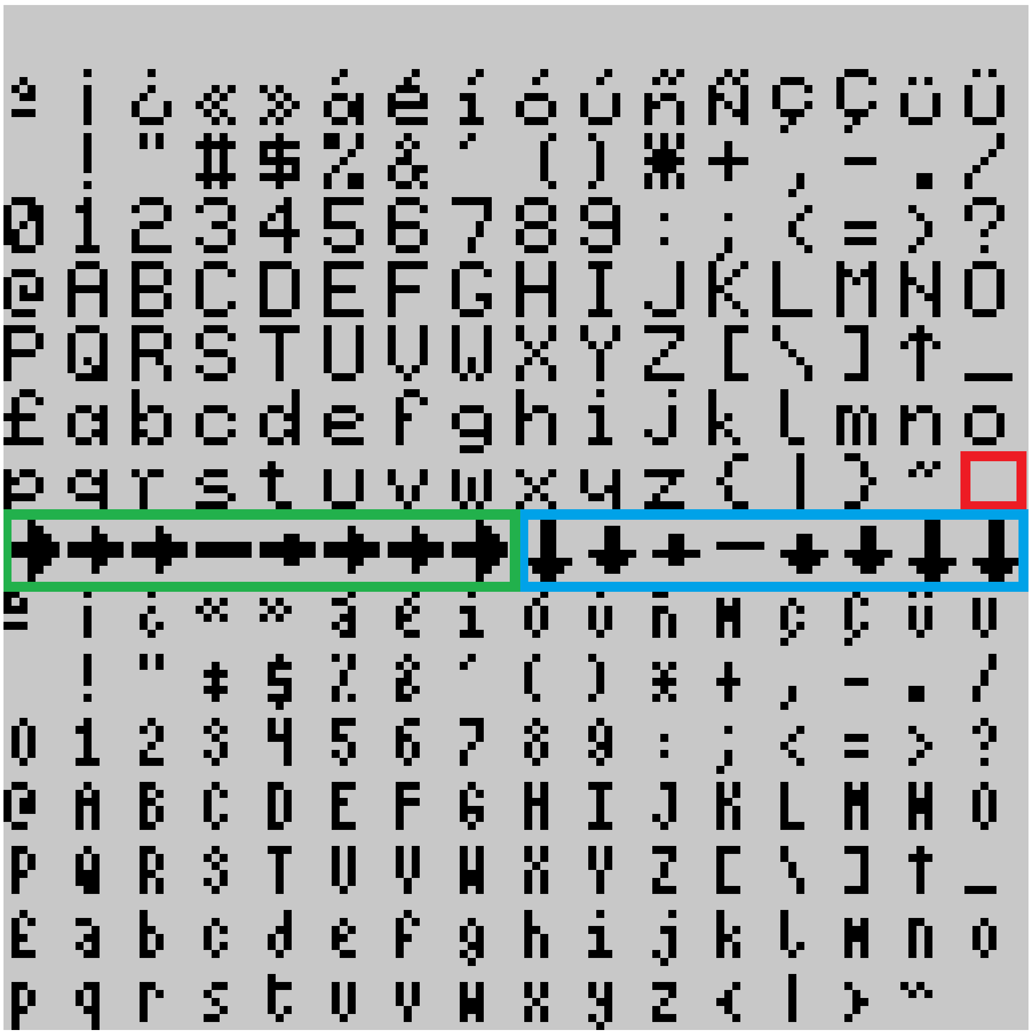
Changing the character set

CYD has the following default character set:



They are arranged in the image as a grid, starting from the top left, and going from left to right and up to down.

You can see both the 6x8 characters of the normal mode and the 4x8 characters of the alternate mode, as well as the characters used in the animated wait and selection icon:

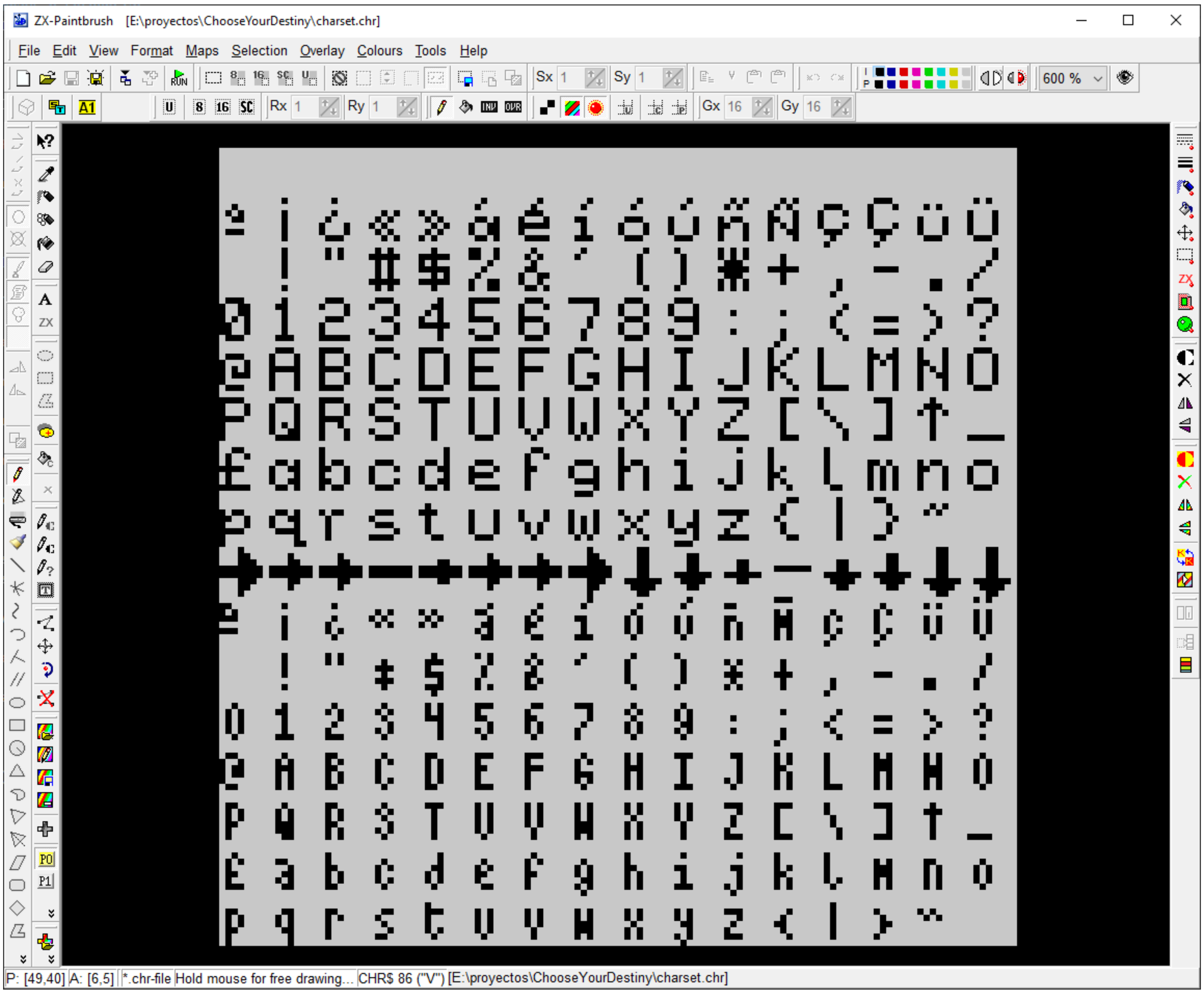


Characters 127 through 143 (both inclusive and starting from zero) are special and have the following functions:

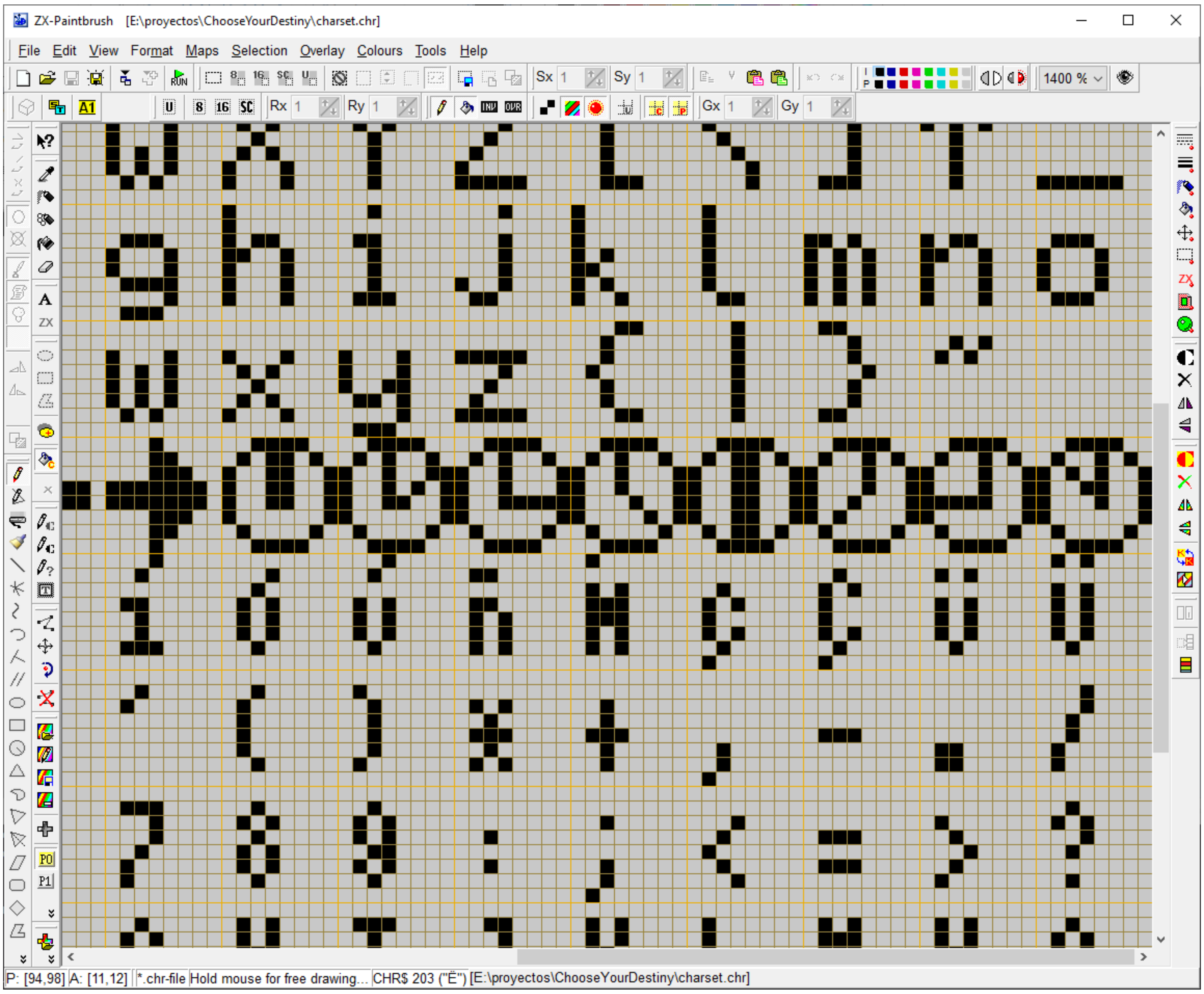
- Character 127 is the character used when an option is not selected in a menu. (In red in the screenshot above)
- Characters 128 through 135 form the animation cycle of a selected option in a menu. (Green in the screenshot above)
- Characters 135 to 143 form the key wait indicator animation cycle. (Blue in the screenshot above)

Once we have this clear, what many of you will want is to change the animated icons to give them a personal touch. For this task we will need **ZxPaintbrush**. There is a copy included in the `Utiles` folder. We will also need the default font found in the `assets\default_charset.chr` file.

We will copy the file to the top directory, rename it as `charset.chr` and open it with ZxPaintbrush and with this we will have this:

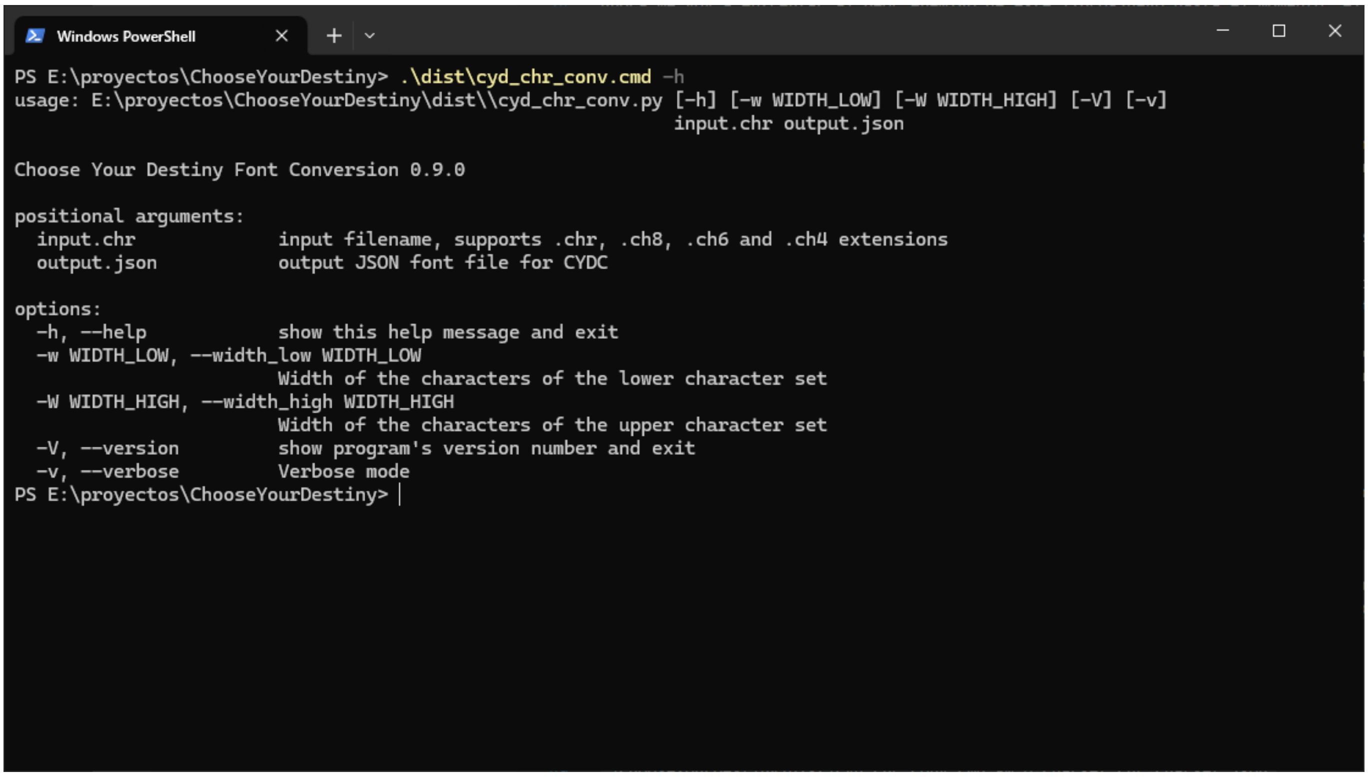

ZX-Paintbrush [E:\proyectos\ChooseYourDestiny\charset.chr]
File Edit View Format Maps Selection Overlay Colours Tools Help
Sx 1 Sy 1 600 %
Rx 1 Ry 1 Gx 16 Gy 16
P: [49,40] A: [6,5] *.chr-file Hold mouse for free drawing... CHR$ 86 ("V") [E:\proyectos\ChooseYourDestiny\charset.chr]

We are going to edit the key wait animation cycle (those marked in blue in the second screenshot). I'm going to put a clock, please excuse my poor artistic skills:


ZX-Paintbrush [E:\proyectos\ChooseYourDestiny\charset.chr]
File Edit View Format Maps Selection Overlay Colours Tools Help
Sx 1 Sy 1 Rx 1 Ry 1 Gx 1 Gy 1
1400 %

P: [94,98] A: [11,12] *.chr-file Hold mouse for free drawing... CHR$ 203 ("Ë") [E:\proyectos\ChooseYourDestiny\charset.chr]

Now we need to convert them to a format that the compiler can digest. To do this we use the cyd_char_conv tool, which is located in `.\dist\cyd_chr_conv.cmd`. This tool is command line, so we will have to open a *Command Prompt* window in the project directory. If we run this command `.\dist\cyd_chr_conv.cmd --help`, we can see the options.



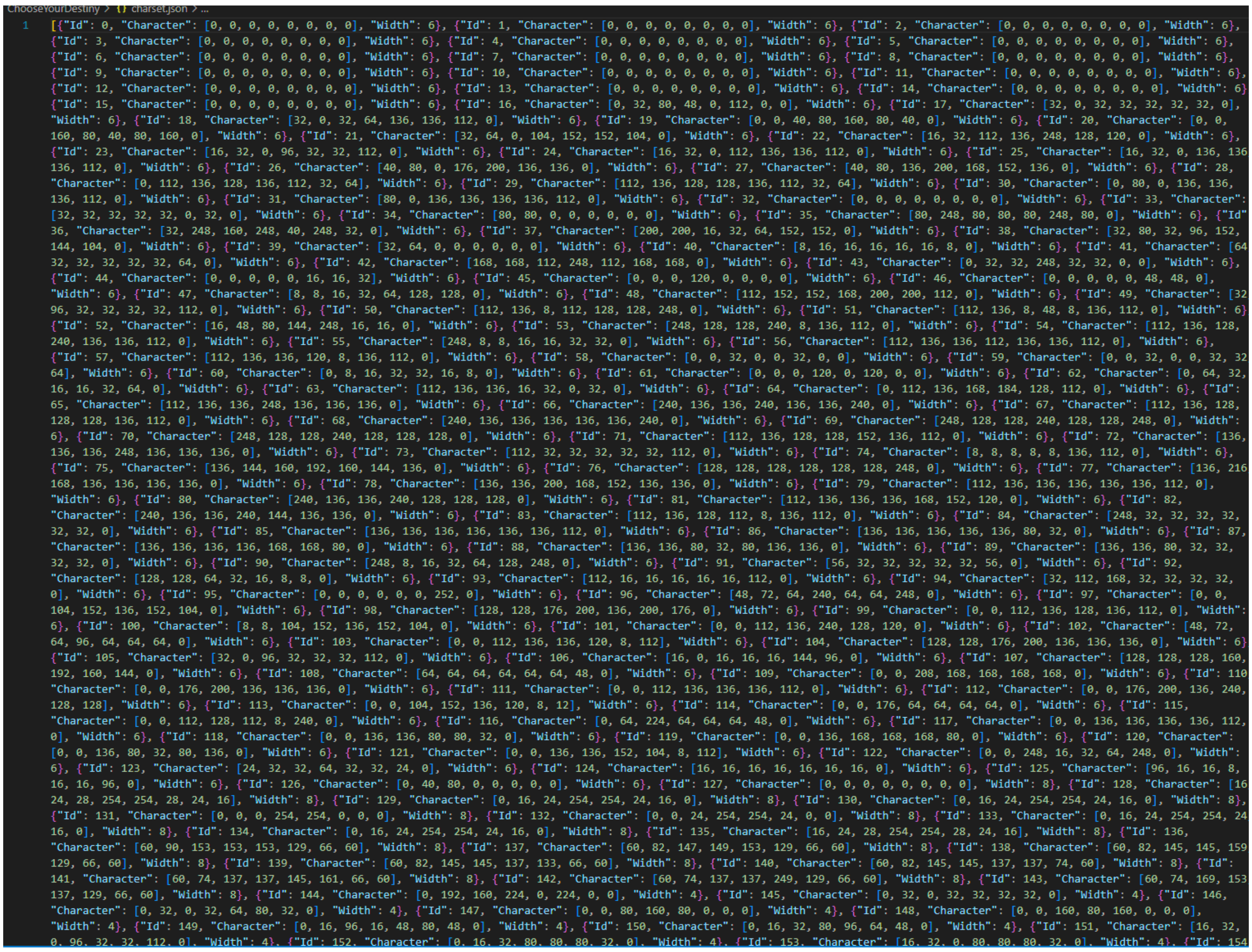
```
PS E:\proyectos\ChooseYourDestiny> .\dist\cyd_chr_conv.cmd -h
usage: E:\proyectos\ChooseYourDestiny\dist\\cyd_chr_conv.py [-h] [-w WIDTH_LOW] [-W WIDTH_HIGH] [-V] [-v]
                                                             input.chr output.json

Choose Your Destiny Font Conversion 0.9.0

positional arguments:
  input.chr             input filename, supports .chr, .ch8, .ch6 and .ch4 extensions
  output.json           output JSON font file for CYDC

options:
  -h, --help            show this help message and exit
  -w WIDTH_LOW, --width_low WIDTH_LOW
                        Width of the characters of the lower character set
  -W WIDTH_HIGH, --width_high WIDTH_HIGH
                        Width of the characters of the upper character set
  -V, --version         show program's version number and exit
  -v, --verbose         Verbose mode
PS E:\proyectos\ChooseYourDestiny> |
```

To do the conversion, we'll use it like this:

```
.\dist\cyd_chr_conv.cmd -w 6 -W 4 charset.chr charset.json
```

With the parameter `-w`, we indicate the width that the characters of the lower set will have, which in our case are 6 and with `-W` we indicate the size of the characters of the upper set, 4 in this case.

With this, we already have the file `charset.json` that we would have to pass to the compiler.



```
[{"Id": 0, "Character": [0, 0, 0, 0, 0, 0, 0, 0], "Width": 6}, {"Id": 1, "Character": [0, 0, 0, 0, 0, 0, 0, 0], "Width": 6}, {"Id": 2, "Character": [0, 0, 0, 0, 0, 0, 0, 0], "Width": 6},
{"Id": 3, "Character": [0, 0, 0, 0, 0, 0, 0, 0], "Width": 6}, {"Id": 4, "Character": [0, 0, 0, 0, 0, 0, 0, 0], "Width": 6}, {"Id": 5, "Character": [0, 0, 0, 0, 0, 0, 0, 0], "Width": 6},
{"Id": 6, "Character": [0, 0, 0, 0, 0, 0, 0, 0], "Width": 6}, {"Id": 7, "Character": [0, 0, 0, 0, 0, 0, 0, 0], "Width": 6}, {"Id": 8, "Character": [0, 0, 0, 0, 0, 0, 0, 0], "Width": 6},
{"Id": 9, "Character": [0, 0, 0, 0, 0, 0, 0, 0], "Width": 6}, {"Id": 10, "Character": [0, 0, 0, 0, 0, 0, 0, 0], "Width": 6}, {"Id": 11, "Character": [0, 0, 0, 0, 0, 0, 0, 0], "Width": 6},
{"Id": 12, "Character": [0, 0, 0, 0, 0, 0, 0, 0], "Width": 6}, {"Id": 13, "Character": [0, 0, 0, 0, 0, 0, 0, 0], "Width": 6}, {"Id": 14, "Character": [0, 0, 0, 0, 0, 0, 0, 0], "Width": 6}
{"Id": 15, "Character": [0, 0, 0, 0, 0, 0, 0, 0], "Width": 6}, {"Id": 16, "Character": [0, 32, 80, 48, 0, 112, 0, 0], "Width": 6}, {"Id": 17, "Character": [32, 0, 32, 32, 32, 32, 32, 0],
"Width": 6}, {"Id": 18, "Character": [32, 0, 32, 64, 136, 136, 112, 0], "Width": 6}, {"Id": 19, "Character": [0, 0, 40, 80, 160, 80, 40, 0], "Width": 6}, {"Id": 20, "Character": [0, 0,
160, 80, 40, 80, 160, 0], "Width": 6}, {"Id": 21, "Character": [32, 64, 0, 104, 152, 152, 104, 0], "Width": 6}, {"Id": 22, "Character": [16, 32, 112, 136, 248, 128, 120, 0], "Width": 6},
{"Id": 23, "Character": [16, 32, 0, 96, 32, 32, 112, 0], "Width": 6}, {"Id": 24, "Character": [16, 32, 0, 112, 136, 136, 112, 0], "Width": 6}, {"Id": 25, "Character": [16, 32, 0, 136, 136
136, 112, 0], "Width": 6}, {"Id": 26, "Character": [40, 80, 0, 176, 200, 136, 136, 0], "Width": 6}, {"Id": 27, "Character": [40, 80, 136, 200, 168, 152, 136, 0], "Width": 6}, {"Id": 28,
"Character": [0, 112, 136, 128, 136, 112, 32, 64], "Width": 6}, {"Id": 29, "Character": [112, 136, 128, 128, 136, 112, 32, 64], "Width": 6}, {"Id": 30, "Character": [0, 80, 0, 136, 136,
136, 112, 0], "Width": 6}, {"Id": 31, "Character": [80, 0, 136, 136, 136, 136, 112, 0], "Width": 6}, {"Id": 32, "Character": [0, 0, 0, 0, 0, 0, 0, 0], "Width": 6}, {"Id": 33, "Character":
[32, 32, 32, 32, 32, 0, 32, 0], "Width": 6}, {"Id": 34, "Character": [80, 80, 0, 0, 0, 0, 0, 0], "Width": 6}, {"Id": 35, "Character": [80, 248, 80, 80, 80, 248, 80, 0], "Width": 6}, {"Id"
36, "Character": [32, 248, 160, 248, 40, 248, 32, 0], "Width": 6}, {"Id": 37, "Character": [200, 200, 16, 32, 64, 152, 152, 0], "Width": 6}, {"Id": 38, "Character": [32, 80, 32, 96, 152,
144, 104, 0], "Width": 6}, {"Id": 39, "Character": [32, 64, 0, 0, 0, 0, 0, 0], "Width": 6}, {"Id": 40, "Character": [8, 16, 16, 16, 16, 16, 8, 0], "Width": 6}, {"Id": 41, "Character": [64
32, 32, 32, 32, 32, 64, 0], "Width": 6}, {"Id": 42, "Character": [168, 168, 112, 248, 112, 168, 168, 0], "Width": 6}, {"Id": 43, "Character": [0, 32, 32, 248, 32, 32, 0, 0], "Width": 6},
{"Id": 44, "Character": [0, 0, 0, 0, 0, 16, 16, 32], "Width": 6}, {"Id": 45, "Character": [0, 0, 0, 120, 0, 0, 0, 0], "Width": 6}, {"Id": 46, "Character": [0, 0, 0, 0, 0, 48, 48, 0],
"Width": 6}, {"Id": 47, "Character": [8, 8, 16, 32, 64, 128, 128, 0], "Width": 6}, {"Id": 48, "Character": [112, 152, 152, 168, 200, 200, 112, 0], "Width": 6}, {"Id": 49, "Character": [32
96, 32, 32, 32, 32, 112, 0], "Width": 6}, {"Id": 50, "Character": [112, 136, 8, 112, 128, 128, 248, 0], "Width": 6}, {"Id": 51, "Character": [112, 136, 8, 48, 8, 136, 112, 0], "Width": 6}
{"Id": 52, "Character": [16, 48, 80, 144, 248, 16, 16, 0], "Width": 6}, {"Id": 53, "Character": [248, 128, 128, 240, 8, 136, 112, 0], "Width": 6}, {"Id": 54, "Character": [112, 136, 128,
240, 136, 136, 112, 0], "Width": 6}, {"Id": 55, "Character": [248, 8, 8, 16, 16, 32, 32, 0], "Width": 6}, {"Id": 56, "Character": [112, 136, 136, 112, 136, 136, 112, 0], "Width": 6},
{"Id": 57, "Character": [112, 136, 136, 120, 8, 136, 112, 0], "Width": 6}, {"Id": 58, "Character": [0, 0, 32, 0, 0, 32, 0, 0], "Width": 6}, {"Id": 59, "Character": [0, 0, 32, 0, 0, 32, 32
64], "Width": 6}, {"Id": 60, "Character": [0, 8, 16, 32, 32, 16, 8, 0], "Width": 6}, {"Id": 61, "Character": [0, 0, 0, 120, 0, 120, 0, 0], "Width": 6}, {"Id": 62, "Character": [0, 64, 32,
16, 16, 32, 64, 0], "Width": 6}, {"Id": 63, "Character": [112, 136, 136, 16, 32, 0, 32, 0], "Width": 6}, {"Id": 64, "Character": [0, 112, 136, 168, 184, 128, 112, 0], "Width": 6}, {"Id":
65, "Character": [112, 136, 136, 248, 136, 136, 136, 0], "Width": 6}, {"Id": 66, "Character": [240, 136, 136, 240, 136, 136, 240, 0], "Width": 6}, {"Id": 67, "Character": [112, 136, 128,
128, 128, 136, 112, 0], "Width": 6}, {"Id": 68, "Character": [240, 136, 136, 136, 136, 136, 240, 0], "Width": 6}, {"Id": 69, "Character": [248, 128, 128, 240, 128, 128, 248, 0], "Width":
6}, {"Id": 70, "Character": [248, 128, 128, 240, 128, 128, 128, 0], "Width": 6}, {"Id": 71, "Character": [112, 136, 128, 128, 152, 136, 112, 0], "Width": 6}, {"Id": 72, "Character": [136,
136, 136, 248, 136, 136, 136, 0], "Width": 6}, {"Id": 73, "Character": [112, 32, 32, 32, 32, 32, 112, 0], "Width": 6}, {"Id": 74, "Character": [8, 8, 8, 8, 8, 136, 112, 0], "Width": 6},
{"Id": 75, "Character": [136, 144, 160, 192, 160, 144, 136, 0], "Width": 6}, {"Id": 76, "Character": [128, 128, 128, 128, 128, 128, 248, 0], "Width": 6}, {"Id": 77, "Character": [136, 216
168, 136, 136, 136, 136, 0], "Width": 6}, {"Id": 78, "Character": [136, 136, 200, 168, 152, 136, 136, 0], "Width": 6}, {"Id": 79, "Character": [112, 136, 136, 136, 136, 136, 112, 0],
"Width": 6}, {"Id": 80, "Character": [240, 136, 136, 240, 128, 128, 128, 0], "Width": 6}, {"Id": 81, "Character": [112, 136, 136, 136, 168, 152, 120, 0], "Width": 6}, {"Id": 82,
"Character": [240, 136, 136, 240, 144, 136, 136, 0], "Width": 6}, {"Id": 83, "Character": [112, 136, 128, 112, 8, 136, 112, 0], "Width": 6}, {"Id": 84, "Character": [248, 32, 32, 32, 32,
32, 32, 0], "Width": 6}, {"Id": 85, "Character": [136, 136, 136, 136, 136, 136, 112, 0], "Width": 6}, {"Id": 86, "Character": [136, 136, 136, 136, 136, 80, 32, 0], "Width": 6}, {"Id": 87,
"Character": [136, 136, 136, 136, 168, 168, 80, 0], "Width": 6}, {"Id": 88, "Character": [136, 136, 80, 32, 80, 136, 136, 0], "Width": 6}, {"Id": 89, "Character": [136, 136, 80, 32, 32,
32, 32, 0], "Width": 6}, {"Id": 90, "Character": [248, 8, 16, 32, 64, 128, 248, 0], "Width": 6}, {"Id": 91, "Character": [56, 32, 32, 32, 32, 32, 56, 0], "Width": 6}, {"Id": 92,
"Character": [128, 128, 64, 32, 16, 8, 8, 0], "Width": 6}, {"Id": 93, "Character": [112, 16, 16, 16, 16, 16, 112, 0], "Width": 6}, {"Id": 94, "Character": [32, 112, 168, 32, 32, 32, 32,
0], "Width": 6}, {"Id": 95, "Character": [0, 0, 0, 0, 0, 0, 252, 0], "Width": 6}, {"Id": 96, "Character": [48, 72, 64, 240, 64, 64, 248, 0], "Width": 6}, {"Id": 97, "Character": [0, 0,
104, 152, 136, 152, 104, 0], "Width": 6}, {"Id": 98, "Character": [128, 128, 176, 200, 136, 200, 176, 0], "Width": 6}, {"Id": 99, "Character": [0, 0, 112, 136, 128, 136, 112, 0], "Width":
6}, {"Id": 100, "Character": [8, 8, 104, 152, 136, 152, 104, 0], "Width": 6}, {"Id": 101, "Character": [0, 0, 112, 136, 240, 128, 120, 0], "Width": 6}, {"Id": 102, "Character": [48, 72,
64, 96, 64, 64, 64, 0], "Width": 6}, {"Id": 103, "Character": [0, 0, 112, 136, 136, 120, 8, 112], "Width": 6}, {"Id": 104, "Character": [128, 128, 176, 200, 136, 136, 136, 0], "Width": 6}
{"Id": 105, "Character": [32, 0, 96, 32, 32, 32, 112, 0], "Width": 6}, {"Id": 106, "Character": [16, 0, 16, 16, 16, 144, 96, 0], "Width": 6}, {"Id": 107, "Character": [128, 128, 128, 160,
192, 160, 144, 0], "Width": 6}, {"Id": 108, "Character": [64, 64, 64, 64, 64, 64, 48, 0], "Width": 6}, {"Id": 109, "Character": [0, 0, 208, 168, 168, 168, 168, 0], "Width": 6}, {"Id": 110
"Character": [0, 0, 176, 200, 136, 136, 136, 0], "Width": 6}, {"Id": 111, "Character": [0, 0, 112, 136, 136, 136, 112, 0], "Width": 6}, {"Id": 112, "Character": [0, 0, 176, 200, 136, 240,
128, 128], "Width": 6}, {"Id": 113, "Character": [0, 0, 104, 152, 136, 120, 8, 12], "Width": 6}, {"Id": 114, "Character": [0, 0, 176, 64, 64, 64, 64, 0], "Width": 6}, {"Id": 115,
"Character": [0, 0, 112, 128, 112, 8, 240, 0], "Width": 6}, {"Id": 116, "Character": [0, 64, 224, 64, 64, 64, 48, 0], "Width": 6}, {"Id": 117, "Character": [0, 0, 136, 136, 136, 136, 112,
0], "Width": 6}, {"Id": 118, "Character": [0, 0, 136, 136, 80, 80, 32, 0], "Width": 6}, {"Id": 119, "Character": [0, 0, 136, 168, 168, 168, 80, 0], "Width": 6}, {"Id": 120, "Character":
[0, 0, 136, 80, 32, 80, 136, 0], "Width": 6}, {"Id": 121, "Character": [0, 0, 136, 136, 152, 104, 8, 112], "Width": 6}, {"Id": 122, "Character": [0, 0, 248, 16, 32, 64, 248, 0], "Width":
6}, {"Id": 123, "Character": [24, 32, 32, 64, 32, 32, 24, 0], "Width": 6}, {"Id": 124, "Character": [16, 16, 16, 16, 16, 16, 16, 0], "Width": 6}, {"Id": 125, "Character": [96, 16, 16, 8,
16, 16, 96, 0], "Width": 6}, {"Id": 126, "Character": [0, 40, 80, 0, 0, 0, 0, 0], "Width": 6}, {"Id": 127, "Character": [0, 0, 0, 0, 0, 0, 0, 0], "Width": 8}, {"Id": 128, "Character": [16
24, 28, 254, 254, 28, 24, 16], "Width": 8}, {"Id": 129, "Character": [0, 16, 24, 254, 254, 24, 16, 0], "Width": 8}, {"Id": 130, "Character": [0, 16, 24, 254, 254, 24, 16, 0], "Width": 8},
{"Id": 131, "Character": [0, 0, 0, 254, 254, 0, 0, 0], "Width": 8}, {"Id": 132, "Character": [0, 0, 24, 254, 254, 24, 0, 0], "Width": 8}, {"Id": 133, "Character": [0, 16, 24, 254, 254, 24
16, 0], "Width": 8}, {"Id": 134, "Character": [0, 16, 24, 254, 254, 24, 16, 0], "Width": 8}, {"Id": 135, "Character": [16, 24, 28, 254, 254, 28, 24, 16], "Width": 8}, {"Id": 136,
"Character": [60, 90, 153, 153, 153, 129, 66, 60], "Width": 8}, {"Id": 137, "Character": [60, 82, 147, 149, 153, 129, 66, 60], "Width": 8}, {"Id": 138, "Character": [60, 82, 145, 145, 159
129, 66, 60], "Width": 8}, {"Id": 139, "Character": [60, 82, 145, 145, 137, 133, 66, 60], "Width": 8}, {"Id": 140, "Character": [60, 82, 145, 145, 137, 137, 74, 60], "Width": 8}, {"Id":
141, "Character": [60, 74, 137, 137, 145, 161, 66, 60], "Width": 8}, {"Id": 142, "Character": [60, 74, 137, 137, 249, 129, 66, 60], "Width": 8}, {"Id": 143, "Character": [60, 74, 169, 153
137, 129, 66, 60], "Width": 8}, {"Id": 144, "Character": [0, 192, 160, 224, 0, 224, 0, 0], "Width": 4}, {"Id": 145, "Character": [0, 32, 0, 32, 32, 32, 32, 0], "Width": 4}, {"Id": 146,
"Character": [0, 32, 0, 32, 64, 80, 32, 0], "Width": 4}, {"Id": 147, "Character": [0, 0, 80, 160, 80, 0, 0, 0], "Width": 4}, {"Id": 148, "Character": [0, 0, 160, 80, 160, 0, 0, 0],
"Width": 4}, {"Id": 149, "Character": [0, 16, 96, 16, 48, 80, 48, 0], "Width": 4}, {"Id": 150, "Character": [0, 16, 32, 80, 96, 64, 48, 0], "Width": 4}, {"Id": 151, "Character": [16, 32,
0, 96, 32, 32, 112, 0], "Width": 4}, {"Id": 152, "Character": [0, 16, 32, 80, 80, 80, 32, 0], "Width": 4}, {"Id": 153, "Character": [16, 32, 0, 80, 80, 80, 32, 0], "Width": 4}, {"Id": 154
```

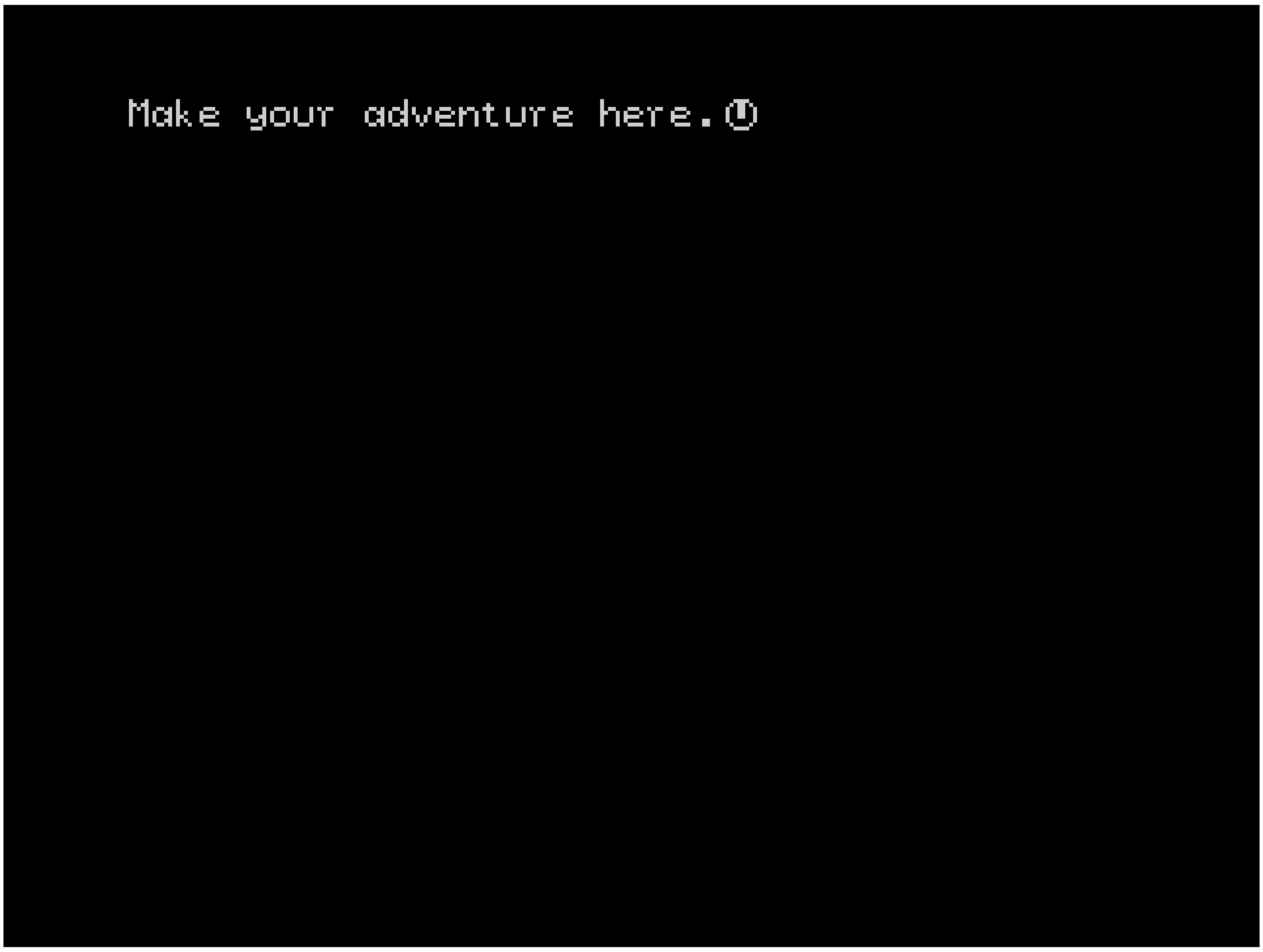
To tell the compiler that we are going to use the new character set, we have to indicate it with the parameter `-c`. Fortunately, the file `make_adv.cmd` will do this work for us.

To do this, we modify the header of that file, putting `-c charset.json` inside the `CYDC_EXTRA_PARAMS` variable, like this:

```
REM ---- Configuration variables ----------

REM Name of the game
SET GAME=test
REM This name will be used as:
REM   - The file to compile will be test.cyd with this example
REM   - The name of the TAP file or +3 disk image

REM Target for the compiler (48k, 128k for TAP, plus3 for DSK)
SET TARGET=48k

REM Number of lines used on SCR files at compressing
SET IMGLINES=192

REM Loading screen
SET LOAD_SCR="LOAD.scr"

REM Parameters for compiler
```

```
SET CYDC_EXTRA_PARAMS=-c charset.json

REM -------------------------------------
...
```

Running the `make_adv.cmd` file and launching the resulting tape file with an emulator, we see that the icon has changed:



```
Make your adventure here.
```

In this way, we can change the rest of the animated icons, the fonts in general or even make special graphic characters, like "UDG", and display them with the `CHAR` command.
