HOW TO HACK AND TRANSLATE ZX SPECTRUM BASIC TEXT ADVENTURES WITH ROM HACKING TECHNIQUES

CONTENTS

INTRODUCTION

The ZX Spectrum was one of the most iconic 8-bit home computers of the 1980s, especially in Europe. Created by Sinclair Research, it became a beloved platform for games, especially early text adventures written in Sinclair BASIC – the built-in programming language stored in the computer's ROM.

Unlike systems that required loading a separate BASIC interpreter, the Spectrum had its BASIC interpreter embedded in ROM, which means games written in BASIC could run directly after loading. This also means that most of the game's logic and text are not compiled into machine code, but exist as editable BASIC listings – though often tokenized or obfuscated to save space.

This guide focuses on text-only adventure games created with BASIC – typically released as .TAP, .TZX, or .DSK files. These are tape or floppy disk images that simulate the original cassette or floppy format the games were distributed on. Once extracted or loaded into an emulator, these files can be edited using ROM hacking techniques, even though the source is not compiled machine code.

We'll explore how to identify and decode BASIC programs, edit their text, change their font for non-Latin alphabets, and even modify the logic and vocabulary of the game itself. Whether your goal is to translate, modify, or simply understand how these old adventures work, this tutorial will guide you step by step through the process.

REQUIRED TOOLS AND SOFTWARE

To hack and translate ZX Spectrum BASIC text adventures, you'll need a combination of general-purpose and Spectrum-specific tools. Here's a breakdown of the most useful ones:

1. You'll need a reliable **ZX Spectrum emulator** to load, test, and sometimes extract game data. The following are popular choices. These emulators also support loading .TAP, .TZX, and .DSK files, making them essential for debugging and testing your translated games. The ability to change game speed (e.g., fast-forward) can also save valuable time.

- **Fuse (Free Unix Spectrum Emulator)**
  One of the most accurate and actively developed Spectrum emulators. Available on Windows, Linux, and macOS.

- **Speccy**
  A versatile and easy-to-use emulator for quickly loading and testing your game. Ideal for those who want a simple solution without the need for installation.

- **Spectaculator (Windows, commercial)**
  Highly polished emulator with excellent support for snapshots and peripheral devices.

2. To be able to edit the bytes of the game file, you need to use a **hex editor**. Any hex editor would do the job, in case you need one to support tables, there are some that are focused on ROM hacking, that do:

- **HxD**
  A fast, lightweight, high-quality and reliable hex editor. Perfect for editing .tap, .tzx, or .dsk files directly.

- **Thingy32**
  Useful when you need support for custom encoding tables (for example, to view text using Greek or other non-standard character sets).

3. If the game uses a custom font (instead of the standard ROM font), it can usually be edited as bitmap graphics:

- **Tile Molester**
  A classic tile editor, suitable for editing Spectrum fonts stored as bitmap data.

4. If you want to edit the loading screen, you will need to use a **graphics editor**:

- **ZX-Paintbrush**
  An excellent tool for editing Spectrum loading screens (SCR format) in native resolution and palette.

- **ZX-Paint**
  A lightweight alternative to ZX-Paintbrush.

5. These editors and emulators are helpful for inspecting and editing BASIC code and blocks:

- **BasinC**
  A modern editor and emulator for Sinclair BASIC with syntax highlighting and testing features.

- **ZX-Blockeditor**
  Allows advanced editing of TZX and TAP files, including BASIC blocks.

- **ZX-Editor**
  Another editor for TAP files with visual editing capabilities, combined with ZX-Blockeditor.

- **Tapir**
  A TAP/TZX file editor with a visual interface, ideal for reordering or repairing blocks, and fixing checksums.

6. If you want to recalculate the checksums, there are at least two programs that take care of it:

- **Tapir**
  It shows the correct checksum, so you can fix it with the hex editor.

- **UpdateTapChecksums**
  Automatically updates the checksums of .TAP files.

7. For performing hexadecimal numbers calculations:

- **Windows Calculator**
  Fast, reliable, and already available.

8. Finally, a text editor is highly recommended:

- **Notepad++**
  A great general-purpose editor to use for scratch work, notes, encoding tables, and text cleanup.

IDENTIFYING AND CONFIRMING BASIC

Before editing or translating a game, you need to be sure that the part of the code you're working with is indeed written in Sinclair BASIC. Many ZX Spectrum text adventures store at least part of their logic and text in BASIC format.
There are a few ways to identify BASIC data within a file:

1. Visual identification using a hex editor

If you're somewhat experienced, you can often recognize BASIC by simply opening the file in a hex editor:
Look for the byte 0D, which marks the beginning of each BASIC token (e.g., PRINT, INPUT, IF).
This byte is followed by a token code and then either arguments (text, numbers) or more tokens.
The start of a BASIC program usually contains many such 0D tokens at regular intervals.
You may also spot line numbers stored as 2 bytes (big-endian), followed by a 2-byte line length, and then the tokenized code.
A few repeating lines with 0D-prefixed tokens are usually enough to confirm it.

2. Using emulators or editors

Tools like BasinC, ZX-Blockeditor, or Tapir can help identify BASIC blocks:
They parse the TAP or TZX file and highlight where the BASIC program begins.
If you can see tokenized lines, you're dealing with Sinclair BASIC.

3. Structure of a BASIC line (for manual inspection)

Each BASIC line in memory has this structure:

Bytes	Description
2 bytes	Line number (big-endian)
2 bytes	Line length (in bytes)
N bytes	Tokenized BASIC code
0x0D	End-of-line marker

This makes it easier to visually confirm whether what you're looking at is BASIC or just ASCII/graphic/text data.
The 0D byte may appear in other parts of the file as well (e.g., in graphic data or screen memory), so it's not a guaranteed identifier on its own. However, if you see a few repeating patterns with valid tokens and line numbers, it's a strong indication.

CHANGING THE FONT

Some ZX Spectrum text adventures use a custom font, while others rely entirely on the built-in system font. If you're planning to translate or localize a game that includes accented characters or a different script, modifying the font is often necessary – but only possible if the game actually includes its own font data.

1. Does the game use a custom font?

To determine whether a game includes its own font:

Run the game in an emulator.
Compare the text on screen to the standard ZX Spectrum system font.
If the letters look different in shape or spacing, the game most likely uses a custom font embedded in the file.

However, keep in mind that some games may use a custom font that looks identical to the system font. In such cases, you'll need to inspect the file directly.

Note: This tutorial only covers games that already include their own font. Adding a custom font to a game that uses the system font is possible, but it involves deeper modification of the BASIC or machine code, and is beyond the scope of this guide.

2. Finding the font in a tile editor

If you suspect the game includes a custom font, the best way to confirm and edit it is by using a tile editor such as Tile Molester.

Open the game's file in Tile Molester.
Select 1bpp planar codec, as the font only uses two colors.
Scroll through the graphics data until you spot recognizable font glyphs – they often appear as 8x8 pixel tiles.
In some cases, you may be able to see the font, but somewhat distorted, or with mixed parts of characters in the same tile. In that case, try adding an offset to that address. Progressively increase the offset, if needed.

If the font is visible and editable, you can proceed to modify the characters to support your translation (for example, adding accented letters or symbols).

3. Edit the font

After you have found the font in the tile editor, you can start editing it. Keep in mind that the editor shows all bytes of the file as graphics, so you may accidentally edit bytes that don’t belong to the font data. So, don’t forget to keep backups of the file.

Before starting to edit the font, you must consider the order of the letters. In text adventure games, players will likely need to type commands. This means the game expects a specific visual glyph to be shown when a certain key is pressed. So, the font data must match the keymap the game uses. So, the order of the letters is determined by that.

For example, if you have a game in another language and you want to translate it in English, you have to draw letter A on the tile that you see the foreign letter that is shown when you press A on the keyboard.

If the game you are going to translate does not use the keyboard (if it uses the arrow keys and Enter), you can chose whichever order you want, for the letters.

If the game ever displays the entire alphabet in order – for example, on a name entry screen – you may want to preserve alphabetical order, unless you can edit how that screen is rendered.

Depending on the available space, you may decide to have small and capital letters. If the game only shows capital letters, you may want to duplicate the capital letters, having them both on the capital and the small letters. Or, you may want to take advantage of the available space by adding other letters and characters (e.g. support English and another language at the same time).

It all depends on the game and your needs. In any case, these are decisions you must take early in the process, before starting changing the text.

4. Text encoding tables

If you change the order of the letters in the font (as they appear in the tile editor), the byte values that produce each letter will no longer match the standard ASCII codes. In that case, you’ll need to create a custom character table so you can correctly read and edit the game’s text in a hex editor.
This is explained in the next chapter: TEXT ENCODING TABLES.

TEXT ENCODING TABLES

Modifying the font defines what the letters look like – but each character on screen is actually mapped to a specific byte value, based on a character table.

The ZX Spectrum character set is a variant of ASCII, and most BASIC games follow this standard mapping. If you preserve this mapping – even partially – you'll be able to edit the text more easily using a standard hex editor like HxD and take advantage of its features and ease of use.

However, if you decide to change the mapping (for example, to add accented letters or different symbols), you'll need to use a hex editor that supports custom tables, like Thingy32. Otherwise, it will assume standard ASCII values, and you'll end up seeing the wrong letters during editing.

If your modified font uses a different encoding than the original (for example, you replaced the character that used to be C with an accented É), then you'll need to create a custom table file that reflects this new mapping.

During the translation process, you'll often need to work with both the original and the modified encoding – especially if you're translating line by line or preserving parts of the original text.

For this reason, it's a good idea to create two table files: one for the original character mapping and one for your new custom encoding.

Thingy32 allows you to load two tables simultaneously and switch between them to view both the original and translated text correctly. This is especially useful during progressive translation, where source and target text coexist in the same file.

This dual-table setup helps avoid confusion and ensures that you're always editing the correct byte values for each character.

Creating a character table

A character table is a UTF-8 plain text file with a .tbl extension.
You can create it using any simple text editor, such as Notepad or Notepad++.

The format is very simple:
Each line contains a byte value in hexadecimal, followed by an equals sign (=), and the corresponding character.

For example, if the letter A is stored as byte 41 and the letter B as 42, your table should include:

41=A
42=B

How to find the byte values

To build your table, you need to know which byte corresponds to each character. Here's a simple method:

Open the game file in any hex editor that displays ASCII (such as HxD).
Look for recognizable text – like menus or in-game messages. Most games use ASCII or a variation of it, so you may be able to read parts of the original text directly.
Match the visible letters with the hex values shown on the left.

For example, if you see the word START in the ASCII column, and the corresponding hex bytes are:

53 54 41 52 54

then your table should include:

41=A
...
52=R
53=S
54=T

You don't have to find every character this way. Once you identify one letter, it's much faster to use the font in a tile editor:

Find the tile that corresponds to the letter you've already identified.
Characters are typically laid out in order: the next tile will represent the next byte (+1), the previous tile will be the byte before (–1), and so on. This lets you fill in the rest of the table quickly by following the visual order in the tile editor.

You can skip characters that you don't need in your translation, or totally custom characters. Also, if you decide to add special characters that Thingy32 can't display, don't add them to the table. Instead, type their byte value directly when needed – that's a limitation of Thingy32.

Once you've created the table for the original encoding, you can save a copy and modify it to match your new character set, based on the font changes you've made.

EDITING TEXT

Text editing is done using a hex editor.

If the game uses standard ASCII or something close to it, you can work directly with a powerful hex editor like HxD, which offers great features, fast navigation, and a modern interface.

However, if the text uses a custom encoding, you'll need a hex editor that supports character tables – such as Thingy32. In that case, Thingy32 will let you read and edit the text correctly, provided you've loaded the right .tbl file.

Basic editing: Replacing text

In most games, you can directly replace existing text by typing over it in the hex editor. This is the simplest method, but it comes with important limitations:

You can't make the text longer than the original, unless there's free space after it.

If your translated text is shorter, you'll often need to pad it with space characters to fill the original space – which can waste valuable bytes.

Because of these limitations, basic overwrite editing can lead to poor translations, awkward phrasing, or inefficient use of space. It's a useful technique for beginners to experiment and test small changes, but it's not ideal for a full-quality localization.

Be careful when editing bytes

When using a hex editor, you're working directly with the game's data. A small mistake can corrupt the file or break the game. Here are a few things to watch out for:

Don't change unrelated bytes. Make sure you're editing the correct section of the ROM – usually the one with visible text.

Avoid overwriting control bytes unless you understand what they do. Some byte values, like FF, may mark the end of a string, a line break, or a command. If you accidentally insert one into the middle of a sentence, the game may cut off the text or behave unpredictably.

Use overwrite mode – not insert mode. Inserting bytes will shift everything that follows, breaking pointers and structure.

If you're unsure what a particular byte does, leave it alone – or make a backup before changing it.

This chapter covers the basics of direct text editing.
In the next chapter, we’ll explore how to change the length of strings and manage pointers, which will allow for proper translation and expansion of text.

CHANGING STRINGS' LENGTHS

In the previous chapter, we explained how to edit text by replacing it directly without changing its length. However, this approach is very limiting: in many cases, we will want to expand or shorten phrases to adapt their meaning correctly.

To change the length of a string, we first need to understand how the main game code is organized.

Structure of commands within the code

The main block of code consists of a sequence of commands.
Each command has a small header that contains:

- The end marker for the previous command (always the byte 0D).
- The command number (2 bytes).
- The command's length in bytes (2 bytes).

In simple terms, each command starts with a structure like:

0D x1 x2 x3 x4

After the header comes the command itself, including data like text, numbers, or other information.

Note: The very first command in the block does not have a preceding 0D, because there is no previous command. However, all others will.

What we need to do when changing the length of a string

When we change the text inside a command (either to make it longer or shorter), we must also update the length value in its header.
Otherwise, the game will misread data, causing crashes or displaying garbage.

The correct procedure is:

- Locate the command header (look for 0D, then the next 4 bytes).
- Edit the text inside the command as needed.
- Calculate the new length of the command.
- Update the two length bytes in the header to reflect the new size.

Proper strategy for changing strings' lengths

When adjusting the length of a string, we cannot simply expand a command and hope for the best. Space must be available first.

Therefore, a proper strategy is needed:

Select a command that will serve as a "space bank".
(It can be any command, preferably early in the block, because we will revisit it often.)

Free up space, e.g. by shortening other commands and move the empty space into the "space bank" command. (We'll see more ways in the following SPACE SAVING chapter).
This way, we always have a pool of free space ready to use.

For example:

Suppose we have a string like "PRESS A BUTTON".
We want to shorten it to "PRESS A".
However, initially we don't remove the extra characters. Instead, we replace it with "PRESS A " – keeping the 7 extra spaces.
These 7 spaces represent available space that we can transfer into the "space bank" command.

Steps:

- Remove the 7 spaces from the original command.
- Update its header length accordingly.
- Add the 7 spaces to the space bank command.
- Update the space bank's header length as well.

This way, the overall block remains consistent, and we gain usable space in a controlled manner.

It's best to accumulate plenty of free space first, before expanding any command.

Maintaining overall balance

The total size of the main code block must be preserved.
If the block size changes (either increasing or decreasing), the program will malfunction or not run at all.

Thus, whenever we adjust a command’s length, we must always balance it by adjusting another command’s length by the opposite amount.

In practice, quality translations usually need more space, because translated phrases tend to be longer.

That’s why it is crucial to:

- Collect spare space early.
- Store it safely in the "space bank".
- Use it later when expanding phrases.

The most obvious way to save space is simply by taking advantage of shorter translated phrases.
However, this is rarely enough: we will need to use various techniques to free even more space.

Before moving on to advanced space-saving tricks, we must first understand BASIC VARIABLES, which we will explore in the next chapter. They can be very helpful for reclaiming space too!

BASIC VARIABLES

In the BASIC language of the ZX Spectrum, there are two types of variables: numeric variables and string variables.

There are also system variables, but these are a completely different topic. System variables are special memory areas used by the machine itself to handle screen coordinates, program pointers, settings, and so on. In this chapter, we will focus only on user-defined variables.

Numeric variables

Numeric variables are used to store numbers, such as counters, scores, or temporary results.
They can have:

One letter as a name (e.g., A, B, X), or two characters (a letter followed by another letter or a digit), like X1, SC, or HP.

In practice, the ZX Spectrum BASIC only considers the first two characters of a variable name significant. Longer names are accepted when typing, but internally only the first two characters are stored and used.

For example:

```
SCORE = 100
SC = SC + 10
```

In the above code, SCORE and SC refer to the same variable, because only SC matters.

String variables

String variables are used to store text, such as words, phrases, or single characters. In Spectrum BASIC, a string variable must consist of exactly one letter plus the dollar sign ($).

Examples:

A$
B$
Z$

You cannot have longer names for string variables, such as NAME$ or HELLO$. These would be invalid.

Because there are only 26 letters in the English alphabet, the maximum number of different string variables you can have at the same time is 26 (A$ through Z$).

Why variables are important

Variables are important because they allow us to reuse values without writing them again and again.
Instead of copying the same word or number multiple times in the program, we can store it once in a variable and refer to it wherever needed.

For example:

```
A$ = "YES"
PRINT A$
PRINT A$
PRINT A$
```

This way, the text "YES" only exists once in memory, even though it is printed three times.

This technique can save a significant amount of space, which is very important when translating or modifying games with limited memory.

In the next chapter, we will see in detail how to use variables strategically to save space and make translations possible even when memory is very tight.

SPACE SAVING

When translating a BASIC program for the ZX Spectrum, memory space is usually very limited. To fit all the translated text into the existing code, we must use various techniques to save as much space as possible, without changing the program’s functionality.

In this chapter, we will learn the most effective ways to save space and ensure our translation fits nicely into the game.

1. Gain from translation

The simplest way to save space is if the translated text is naturally shorter than the original.
For example, if the English phrase "WHAT WILL YOU DO NEXT?" becomes simply "ΤΙ ΘΑ ΚΑΝΕΙΣ;" in Greek, we save several bytes immediately, without any code modification.

2. Removing REM commands

REM commands (used for comments) can be deleted entirely.
On the Spectrum, a REM line uses up memory both for the command itself and for the comment text.

By deleting REM lines, we instantly free up memory.
You can delete the whole command, including the header, not just the comment.

Warning: In theory, a GO TO or GO SUB could jump to a line containing only a REM. In practice, though, this is extremely rare in compiled games. In case it does happen, you must update the GO TO or GO SUB command to go to the following command.

3. Using whitespace

If the original code contains unnecessary spaces within BASIC lines (rare but possible), we can take advantage of them.
In BASIC, spaces inside commands (for example, PRINT "HELLO") are ignored by the interpreter.
Thus, if necessary, we can reuse this empty space for our translated text.

4. Renaming variables

Variable names appear many times throughout the code.
For example, if there is a numeric variable called money, we can rename it to m, saving 4 bytes each time it is used.

This can be done for the numeric variables only, as the string variables are always two characters long, always carefully avoid breaking the logic of the program.

5. Using string variables for common phrases

If a phrase appears many times in the program, we can store it in a string variable and reuse it. For example, you can replace these commands:

```
PRINT "YOU TOOK THE KEY"
PRINT "YOU TOOK THE SWORD"
```

With these:

```
LET A$="YOU TOOK THE "
PRINT A$;"KEY"
PRINT A$;"SWORD"
```

This way, we save space every time the same text is printed.

The total number of string variables is 26 (A$, B$, …, Z$). The original game will have a number of string variables anyway, so you can create some yourself. You have to calculate and think the best way to use any new variables.
Keep in mind that if a long string appears some times in a big part of the game, but you know it’s not used afterwards, you can change its value for the next part.

6. Using numeric variables for frequent numbers

The same idea applies to numbers that are used repeatedly.
For example:

```
LET N=5000
GO TO N
```

Instead of writing the number again and again, we use a variable and save valuable bytes.

7. Smart value assigning in numeric variables

To use any numeric variables, you must assign values to them, at the beginning.

For example, a command that assigns 100 to a variable, takes up 6 bytes (if you use VAL(), see below). If you already have another variable that holds the value 10, you can multiply that one with itself, and that would take up only 3 bytes, saving you 3 bytes.

If you think smart, you can first declare a variable with 1, and then use that trick for 0, 2, 3, etc. Depending on the order of the declarations of the variables, you can apply that trick for more numbers.

8. Combining commands

In BASIC, we can put multiple commands on the same line, separated by ":".
For example:

LET A=5:PRINT A

This avoids the need for multiple line headers, saving space.

9. Using VAL() for numbers

Numbers in BASIC are stored in memory in a format that takes up many bytes.
For example, the number 2500 is stored like this:

32 35 30 30 0E 00 00 C4 09 00

However, the command VAL "2500" stores the number as text ("2500") and converts it into a number during program execution, using only 7 bytes:

B0 22 32 35 30 30 22

In the hex editor, that would look as:

°"2500"

This way, we save 3 bytes each time, with only a tiny impact on execution speed – completely negligible in text adventure games.

10. Changing line numbers

If space becomes really tight, we can reassign some BASIC line numbers, moving them to numbers with fewer digits.

Important notes:

- We must also update all GO TO and GO SUB commands that point to these commands, accordingly.
- Line numbers must continue to increase properly throughout the code.
- This method is very powerful but also risky if not done carefully.

11. Code optimization

If we have enough experience, we can modify the code to do the same job with fewer or different commands, that take up less memory.
This is a more advanced technique and should only be used when all simpler methods have been exhausted.
This will be discussed in chapter EDITING BASIC CODE AND ADDING GAME COMMANDS.

Conclusion

By applying these space-saving techniques, we can free up enough memory to fit our entire translation.
In most cases, it is enough to simply make translations shorter, rename variables, and use common string variables.

More advanced techniques, such as optimizing code or changing line numbers, should be used only when absolutely necessary.

Saving space is essential for a successful BASIC game translation on the ZX Spectrum!

SCREEN LIMITATIONS AND TEXT WRAPPING

When writing or translating a text adventure, you must always remember that the Spectrum's screen is limited: it can only display 32 characters per line and 24 lines in total. The game itself can also have further limitations, like a frame for the text, that fits less characters.

If your text is too long to fit, the computer doesn't just quietly scroll the screen – it shows as much as it can, and then prints "SCROLL ?", waiting for the player to press a key before showing the rest.

This looks very ugly during gameplay, so you must always make sure that your messages fit nicely on screen.

Even more important than the total number of lines, though, is the width of each line. The Spectrum automatically wraps text at the 32-character mark – even if that happens in the middle of a word! This can easily make your text look messy and unprofessional, splitting words across two lines.

To avoid this, you have two options:

Either add some extra spaces before the word that would be split, so it moves cleanly to the next line, or rephrase the sentence to make it fit better.

When dealing with longer messages that span multiple lines, be very careful: if you make a change in one of the early lines, you might mess up the wrapping in all the following ones! Always double-check the entire text after any edit.

A very useful trick is to set your hex editor to display a fixed width of 32 bytes per line. For example, in HxD you can change the view settings so that you see exactly where each screen line would end. Usually, it's best to create a new blank file filled with spaces, and then type your new text over it. This gives you full control over the layout.

If part of the text is replaced by a variable (for example, a name or a number), you need to be extra careful: from the point where the variable appears, the position of everything afterwards can shift depending on the variable's length.

Another thing you must watch out for, is overwriting. Sometimes, a new message is printed over an older one, without first clearing the old text. If your new (translated) text is shorter, remnants of the old message might still be visible. In such cases, you must pad your new text with extra spaces to make sure it fully covers the previous one.

Finally, if you want to insert a blank line during gameplay, you can do it by inserting the bytes F5 27 right (the PRINT command is byte F5), instead of having

PRINT : PRINT

This saves a byte, or more, if you need multiple new lines, as you can have F5 27 27 27...

Remember: in a text adventure, good presentation matters almost as much as saving memory!

EDITING BASIC CODE

Besides translating text and saving memory, you can also modify the actual BASIC code of the game to change how it works or fix bugs!

We have already mentioned that you can:

- Join multiple commands into one line by using ":".
- Change line numbers to save space.

But beyond saving space, you can also add new functionality.
For example, you can:

- Add synonyms for commands or items.
- Add an "ABOUT" or "HELP" page if the game doesn't have one.
- Fix bugs.
- Even introduce new objects or events into the game!

To add or change commands, you must work directly at the byte level.
You don't type commands like PRINT or GO TO in plain text.
Instead, you insert the correct byte value that represents each BASIC command.
For instance, the PRINT command is represented by the byte F5.

Unfortunately, we don't have a full list of all the byte codes for every command.
However, you can easily find the byte for a specific command by looking at the existing game code using a disassembler or a tool like BasinC, ZX-Editor, or Tapir.
If you see a line of code that uses the command you're interested in, you can check its byte value in a hex editor.

You don't even have to find it inside the same game – you can look at any other Spectrum BASIC program to find the byte you need!

Finally, when adding new lines of code, remember that you must also insert the proper header (the line number and length bytes) before the actual line contents, as we explained earlier.

Always remember: you are editing at the byte level! You must insert the proper bytes for each BASIC command, not the text version.
For example, PRINT is byte F5, not the letters P R I N T. But you can always change the file this way and check the BASIC code listing in BasinC, ZX-Editor, or Tapir.

With careful editing, you can really expand what a game can do!

ADDING GAME COMMANDS

Now that you know how to edit BASIC code, you are ready to expand the game by adding new features!

One of the most powerful things you can do is to add new player commands – but that's not all! You can also add new objects, new rooms (screens), or even new descriptions for existing items that originally had none.

Adding new commands

Sometimes you may want to add synonyms for existing commands.
For example, if the game understands LOOK but not SEE, you can create a small code addition, so that typing SEE will act exactly like LOOK.

Tip:
In many classic games, it became popular to allow players to type only the first two letters of a command instead of the full word.
For example, typing LO would work the same as LOOK, or IN would work as INVENTORY.

If the game you are working on does not support this, you can add these two-letter shortcuts yourself! It makes the game much faster and smoother to play.

Warning:
You must check carefully for conflicts.
If two commands share the same first two letters (e.g., LOOK and LOCK), you must decide how to handle it, perhaps by adjusting the shortcuts.

In general, the steps are:

- Find where the game reads the player's input.
- Locate the code that checks what the player typed.
- Add a new check for your synonym or shortcut.

For example:

```
IF A$="SEE" THEN A$="LOOK"
IF A$="LO" THEN A$="LOOK"
```

Or you can create a new block of actions for a brand-new command.

Expanding the world

Beyond commands, you can also add depth to the game in other ways:

- Add new rooms (screens) by expanding the map.
- Add new objects that the player can interact with.
- Add descriptions for "background objects" that were originally only mentioned.

For example:
Sometimes a room's description mentions a statue or a carpet, but if you try to LOOK at it, nothing happens.
You can improve the experience by adding a small description, even if the object is not important to the game itself!
This makes the world feel more alive and detailed.

Expanding a game like this not only improves the player experience but also gives your translation a special personal touch. It's a great way to make your work truly stand out!

CHECKSUM RECALCULATION

When editing a Spectrum TAP or TZX file, there is something very important you must not forget:
Every block of data in the file includes a checksum byte at the end.

The Spectrum uses this checksum to verify that the data was loaded correctly.
If the checksum is wrong, the Spectrum will display an R Tape loading error when trying to load the block.

In many modern emulators, if you enable fast loading, the emulator simply ignores checksum errors.
This is why, during translation work, you may not notice anything wrong – the game keeps loading fine even if the checksum is broken!

However, if you try to load your modified game normally – without fast loading, or on a real ZX Spectrum – a wrong checksum will cause a loading error, and the game will not load.

Because of this, you must recalculate and correct the checksum of the modified block if you want to check the loading screen properly, or when the game is ready to be released.

How to update the checksum

You don’t need to recalculate the checksum manually, there are at least two tools that can do it for you. You can either use the command-line program UpdateTapChecksums to update the checksum (it only supports TAP files), or Tapir, which shows you the correct checksum byte of the selected block, so you can update it yourself.

Fixing the checksum is a simple step, but very important to make sure your translated game is fully compatible – not just with emulators, but also with the real ZX Spectrum!

DEBUGGING AND TROUBLESHOOTING

While translating or modifying a ZX Spectrum game, it's almost inevitable that you will encounter problems.
Understanding the most common mistakes and knowing how to find and fix them is an important skill.

Here are some of the most frequent sources of errors:

Incorrect block length

One of the most common mistakes is incorrectly updating the length of a BASIC or data block in the file header.
Sometimes you might need to subtract or add a number to the existing length, but if you get it wrong, the program can crash or behave strangely.

Inserting instead of replacing

Another frequent mistake is accidentally inserting a byte instead of replacing one. This shifts all following bytes, breaking the structure of the program completely.
If your modified file suddenly has a different total size, that's a good sign that this kind of error may have happened.

Checksum errors

As explained in the previous chapter, forgetting to update the checksum after editing a block will cause an "R Tape loading error" when loading without fast loading enabled.

BASIC syntax errors

If you modify BASIC commands incorrectly – for example, by breaking the structure or syntax – the Spectrum will give a syntax error.
The error message will show you the number of the line and which part (between colons ":") caused the problem.

To help you with debugging, you can use a hex editor to inspect the file manually. However, it’s often easier to use tools like BasinC, Tapir, or ZX-Editor.
These programs not only display the BASIC code but also highlight exactly where an error occurs!

In particular, ZX-Editor has a syntax checking feature that can detect errors that may not cause an immediate crash, but would still prevent the game from working correctly.

Sometimes, while translating, you might even discover bugs in the original game! Since these games are written in BASIC, you are able to fix any mistakes you find. There are no general instructions for this – each bug depends on the specific BASIC code.

Finally, it’s important to remember that you might introduce a bug without noticing it immediately.
If you continue making many changes afterward, it can be very hard to locate the problem.

This is why keeping regular backups is essential.
If a bug appears, you can check your earlier backups to pinpoint exactly when it was introduced, then compare the working and broken versions to locate and fix the error.

(We will discuss backups and testing in more detail in the next chapter.)

BACKING UP AND TESTING

When working on a Spectrum translation, it’s important to back up your files often. Translation work can involve many small changes, and mistakes may not always show up right away.
A backup lets you go back to a safe version if something goes wrong.

It’s a good idea to make a new backup after every major change – for example, after finishing the translation of a screen or a block of text (or even after a minor change). This way, if a bug appears later, you can easily go back and check when it was introduced, as we mentioned earlier.

Testing is just as important.
Don’t wait until you have translated everything to test the game!
You should load the game frequently and check if it still works properly after each set of changes.
This way, if something breaks, you will immediately know which change caused the problem. Also, frequent testing helps you spot text alignment issues early, so you can adjust the command length at once instead of fixing multiple places later.

By backing up often and testing regularly, you can catch mistakes early and save yourself a lot of time and trouble later!

FURTHER RESOURCES

Here are some useful resources, tools, and websites that can help you with Spectrum translation and hacking.

Emulators

- Fuse (Free Unix Spectrum Emulator)
  https://fuse-emulator.sourceforge.net/

- Speccy Emulator (by Marat Fayzullin)
  https://fms.komkon.org/Speccy/

- Spectaculator (commercial)
  https://www.spectaculator.com/

Hex editors

- HxD
  https://mh-nexus.de/en/hxd/

- Thingy32 (ROM hacking hex editor with table support)
  https://www.romhacking.net/utilities/570/

Tile editors

- Tile Molester
  https://www.romhacking.net/utilities/991/
  https://github.com/toruzz/TileMolester

Graphics editors

- ZX-Paintbrush
  https://freestuff.grok.co.uk/zxpaintbrush/
- https://worldofspectrum.net/zx-modules/55/index.html

TAP/TZX editing / BASIC editing

- BasinC
  https://sourceforge.net/projects/basinc/

- ZX-Blockeditor
  https://worldofspectrum.net/zx-modules/46/index.html
  https://worldofspectrum.net/zx-modules/217/index.html

- ZX-Editor
  https://worldofspectrum.net/zx-modules/46/index.html
  https://worldofspectrum.net/zx-modules/211/index.html

- Tapir
  https://www.alessandrogrussu.it/tapir/
  https://spectrumcomputing.co.uk/forums/viewtopic.php?t=5626

Checksum tools

- UpdateTapChecksums
  https://www.greekroms.net/english/download.php
  https://www.romhacking.net/utilities/1709/

General information

- https://en.wikipedia.org/wiki/ZX_Spectrum
- https://en.wikipedia.org/wiki/ZX_Spectrum_character_set
- https://www.romhacking.net/
- https://spectrumcomputing.co.uk/
- https://spectrumcomputing.co.uk/forums/
- https://sinclairzxworld.com/
- https://worldofspectrum.org/ZXBasicManual/
- https://sinclair.wiki.zxnet.co.uk/wiki/TAP_format
- https://web.archive.org/web/20001205205200/http://www.void.demon.nl/TZXformat.html
- https://scratchpad.fandom.com/wiki/Spectrum_emulator_file_formats
- https://chuntey.wordpress.com/

CREDITS

This guide was written to help new translators bring classic ZX Spectrum games to life in other languages.

Special thanks to all the developers who created the emulators, editors, and tools mentioned here – without them, this work would be much harder!

I would also like to thank the Spectrum fan communities for keeping the spirit of the ZX Spectrum alive after all these years.

Happy translating!