Xenoxxx Funtime Show - the technology.

This is written for three groups. Your rank corresponds to how far you can stand reading:

- Curious cheaters.
- Casual tinkerers.
- Paul Dunning.

It's also available as a PDF.

An update

Strictly, this is not the original technology. There have been some behind-the-scenes changes to make modifications easier. Most importantly, better compression, leaving more space for your own words and pictures.

So, this is the **Xenoxxx Optimised Revision** (v0.1). It's supposed to play exactly as before, so please let me know when you inevitably find bugs. Likewise, please send any questions or requests to comments at arbitraryfiles.com.

The original development files, and the original version of this guide, are archived here.

Downloads

First: source code, maps, graphics, and text. Sufficient for curious cheaters, and with a suitable compiler (as detailed later), very casual tinkerers.

Here they are.

Next, my asset conversion software. This will allow you to change the graphics, text, and logic. Proceed with caution, because this will have read and write access to your system. I will not be held responsible for any consequential loss, damage, or goujon incidents.

Backed up your files? Here you go then.

Documents

The **Docs** folder contains these spreadsheets:

- **Bytecode commands** is for reference when writing game logic.
- **Drakelow ZX** lays out the tunnel map and events.
- **Event cell tracker and cheat-sheet** details how to trigger those events.
- **Memory map** is for reference when making major modifications.
- **Variables** is also for reference when writing game logic.

The tunnel map and cheat-sheet are sufficient for, as you might have guessed, cheating. The bytecode commands and variable reference, explained later, will assist moderate tinkering. Skip the memory map if it looks too scary.

Source code

The **Game** folder contains the **Data** subfolder, and a bunch of source code.

XOR.asm is the main code file. All the other **.asm** files, and the contents of the **Data** subfolder, will automatically be included when you compile **XOR.asm**.

Data

The **Data** folder contains:

- **.png** files, images with up to three colours (black, white, magenta.)
- **.scr** files, images in the ZX Spectrum's screen display format.
- **.txt** files, containing game words and logic.
- **.asm** and **.bin** data files, automatically included when compiling **XOR.asm**.

Simple modification

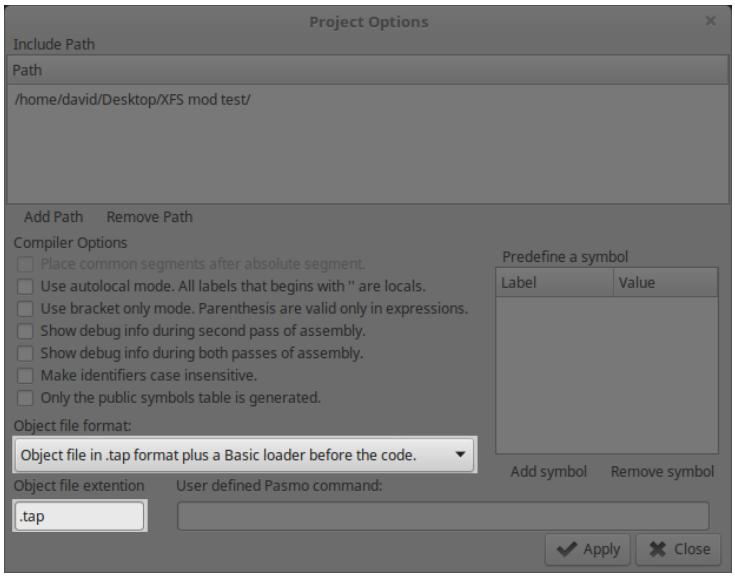
Now, I'll explain how to set up the compiler, build the game file, and run it. Once you've done that, making your first modification is easy.

First, install zDevStudio.

Create a new directory for the game files. Start zDevStudio, then create a new project called “XOR” in that directory. It will automatically create an empty code file for you. Now close zDevStudio.

Copy the contents of the **Game** folder, including the **Data** subfolder, to that directory, overwriting the empty code file that zDevStudio just created.

Open zDevStudio again. Select **Build** then **Build Options** from the menus. You will need to change the **Object file format** to “Object file in .tap format plus a Basic loader before the code.”, change the **Object file extension** to “.tap”, then apply the changes.

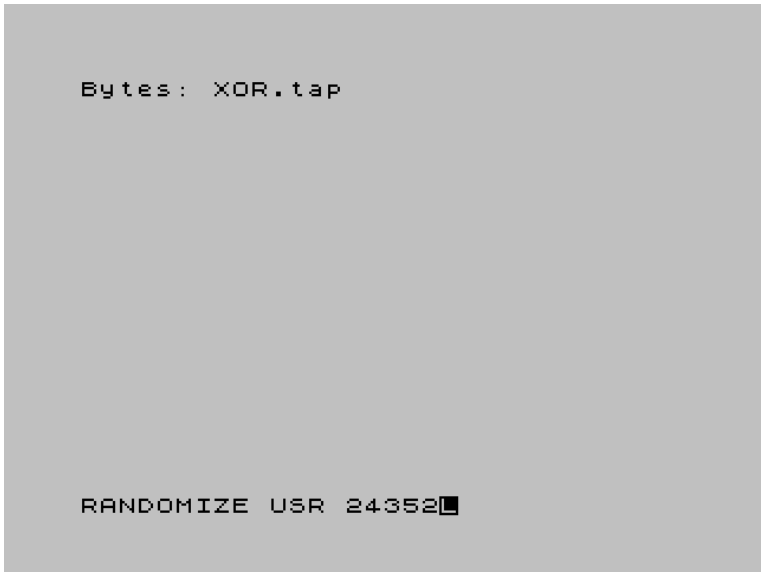


Now, select **Build** then **Build Project** from the menus. Some messages will appear at the bottom window, and **XOR.tap** should appear in the directory you created. This file can be opened with your ZX Spectrum emulator, but the game will not load immediately.

If your emulator has started in 48k mode:

- Press **t,** and RANDOMIZE should appear at the bottom of the screen, with a flashing capital L beside it.
- Hold down **Control** then press **Shift**. The capital L should change to a capital E.
- Press **I**, and the bottom of the screen should now read RANDOMIZE USR.
- Type **24352**, then press **Enter**. The game should then start at the title screen.

Otherwise, type **RANDOMIZE USR 24352** in full.



You've done the hardest part, now to make a simple modification: defacing the title screen.

Go to the **Data** subfolder and open the plain text file **TXLINES.asm**. If you're using Windows, and this file appears to have no carriage returns, then try opening it with Wordpad instead of Notepad.

Scroll down to the 67th line, which should read:

DB "", 158, " IS BOUNTIFUL" ;0

Change **" IS BOUNTIFUL"** to **" IS DEAD - GJ”**, or anything else exactly thirteen characters long, including spaces. The example line would now read:

DB "", 158, " IS DEAD - GJ" ;0

Save and close the file. Open zDevStudio, build the project again, then run the new .tap file in your emulator. If you followed the example, then this title screen should appear:

Compilation software

The other zip file contains software that converts images, text, and instructions to a format that the ZX Spectrum can understand. You must have Python installed to run this software.

Unzip the file to the **Data** folder. It contains the following modules:

- **txtlib.py** and **zxasmlib.py**, library files used by other modules.
- **ZX_Image_Compiler.py**, converts images to compressed binary data.
- **ZX_Node_Compiler.py**, converts logic instructions to bytecode.
- **ZX_SCR_Splitter.py**, splits .scr files into images and attribute data.
- **ZX_Text_Compiler.py**, formats and compresses ASCII text.
- **ZX_Text_Compiler_config.py**, configuration file for the above.

Moderate modification

Defacing title screens is briefly amusing, but perhaps you want do more. Say, add a secret mini-game called “Where's Kenny?”

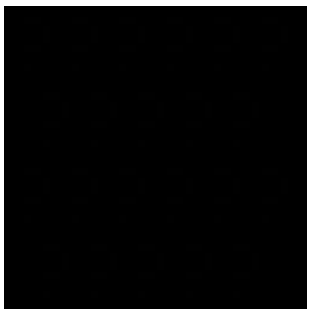
Kenny will be drawn somewhere within a 4x4 grid. One space at a time will be selected randomly, briefly revealed, then disappear again. If the player presses the **5** key when Kenny is visible, they win!

So, this game will need some text, a picture of Kenny, and logical instructions.

Like these, for example.

There are actually two pictures in there: **049.png**, the Kenny, and **050.png**, a background image for each grid square. Solid colours can only be black or white, and magenta indicates transparent pixels. The conversion software automatically determines if each image has an alpha channel, and also complains if it doesn't like the colours.

The dimensions must be a multiple of 8 pixels. Here, they are 32 by 32 pixels, to ensure that the grid fits comfortably on the screen. Within these limitations, you can draw most other types of Kenny (G, Masters, Everett.) Make sure to put them in the **Data** folder.



Now to add some messages. These are stored in a new file called **010.txt**, so that the conversion software will keep them together in line bank 10:

```
WHERE'S KENNY?

Instructions:
Press 5 when Kenny appears. If Kenny is not there, you lose.

Press 1 to start.
Press 2 to quit.

KENNY!
NOT KENNY.
```

Each line is considered as one message, with blank lines indicating an empty space in the sequence. For example, “WHERE'S KENNY?” will be message 0, and “Instructions:” will be message 2.

Then, the logical instructions to bring it all together, which will be explained later. For now, copy **NODES.txt** from the zip file to the **Data** folder, overwriting the original file.

Compiling

Next, converting the images, text, and instructions to a format that the ZX Spectrum can understand. First, check that all the assets and Python programs are in the **Data** folder.

To convert the images, run **ZX_Image_Compiler.py**, then enter **9E00** when prompted for the start address.

For each image, the program stores an array of codes indicating what each 8 by 8 pixel block contains: pure black, pure white, a two byte address pointer, or eight bytes of pixel data. The alpha channel, if present, is processed identically, and stored interleaved with the monochrome pixel blocks.

Finally, all the image data is stored in **BITMAPS.bin**. Each image begins with its array of codes, which is a known byte size in relation to the image dimensions, then a variable-length stream of pointer and pixel data. An index of all the image dimensions, start address, and alpha channel flags is stored in **BITMAPS-I.asm**.

To convert the text, simply run **ZX_Text_Compiler.py**. This will produce four pairs of data and index files, with all index files ending **-I.asm**.

TXDICTWD.asm is generated by searching the text for strings common enough to be worth tokenising. For example, it will now contain both “KENNY” and “Kenny”, because these appear in **010.txt** more than once.

TXTOKENS.asm is essentially a copy of **TXTOKENS.txt**, and tokens are strings that can be substituted within a line. For example, rather than have different lines for each Roamer, the lines are the same, but the roamer name is set by specifying a variable which contains the token number to use.

TXBANKS.asm contains start addresses for the sets of up to 63 lines in each text bank, two of which can be accessed at once. The lines are stored sequentially, so their length can be calculated by subtracting the start address from the start address of the next line. This is why each bank is padded with a final empty entry.

TXLINES.asm contains raw text in quotes and 8 bit numbers, separated by commas:

- **0** is an escape code for tokens, with the following number indicating which variable contains the token number to use.
- **1** to **31** signify a lower case letter followed by a space character.
- **32** to **127** is the range used by non-control ASCII characters. The Z80 compiler automatically converts plain text within quotes to the corresponding numbers.
- **128** to **255** signify dictionary words. The dictionary word number is calculated by subtracting 128.

To convert the instructions, run **ZX_Node_Compiler.py**, then enter **E800** when prompted for the start address. This produces **NODES.bin**, a block of numbers which represents all the instructions and their parameters. It also produces four index files, which break the instructions into groups, ending **I-CUR.bin**, **I-ENT.bin**, **I-EXI.bin**, and **I-SUB.bin**.

If you want to make modifications, then you need to understand those instructions. But first, rebuild the project with zDevStudio, and see if you can find Kenny.

Coding Nodes

This section details the changes that were made in **NODES.txt** to bring Kenny to life. There's much to explain, but at the same time, the changes amount to little more than a single paragraph of additional storage space.

They begin at line 81, where the original instructions were:

```
; Six digits entered?
IFVAR==(NUM) 30, 6
  IFVAR==(NUM) 25, 6
    ; Do shortcut.
    SETLINEBANKHI 4
    SETVAR 42, 5
    SETPGPHAREA 0, 0, 32, 24
    SETPGPHDELAY 5, 5
    SETPGPHSPACE 0
    CHANGENODE 12
  ENDIF
  ; Reset counters.
  SETVAR 25, 0
  SETVAR 30, 0
ENDIF
```

This was part of a check for the key sequence **214214** on the title screen. If the sequence was entered correctly, then **CHANGENODE 12** would cause the game to begin at the tunnel sequence, rather than Cranetron.

Nodes are how instructions are organised, and Node 14 is a new node containing the mini-game. So, the revised instructions are as follows, with the key change highlighted:

```
; Six digits entered?
IFVAR==(NUM) 30, 6
  IFVAR==(NUM) 25, 6
    ; Start the secret mini-game.
    CHANGENODE 14
  ENDIF
  ; Reset counters.
  SETVAR 25, 0
  SETVAR 30, 0
ENDIF
```

Node 14 begins on line 574, as follows. The first example of each instruction is highlighted, and all are documented within **Bytecode commands** in the **Docs** folder:

```
; Where's Kenny?
NODE 14
ONSTATEENTRY
  SETALLATTRIBUTES 64 ; Bright, no flash, Black paper, Black ink.
  FILLSCREEN 0
  SETLINEBANKHI 10
  SETATTRIBUTES 0, 20, 32, 4, 68 ; Bright, no flash, Black paper, Green ink.
  SETPGPHAREA 0, 20, 32, 4
  SETPGPHDELAY 1, 1
  SETPGPHSPACE 0
  SETPGPHMODE 0
  SETVAR 64, 0  ; Kenny's X position.
  SETVAR 65, 0  ; Kenny's Y position.
  SETVAR 66, 0  ; Reveal X position.
  SETVAR 67, 0  ; Reveal Y position.
  SETVAR 68, 0  ; Reveal countdown.
  SETVAR 69, 20 ; How long each reveal lasts.
  SETVAR 70, 5  ; How long the gap between reveals lasts.
  SETVAR 71, 0  ; Flag: reveal is active.
  SETVAR 72, 1  ; Flag: show instructions.
  SETVAR 73, 1  ; Flag: draw Kenny.
  SETVAR 74, 0  ; Flag: game in progress.
  SETVAR 75, 0  ; Grid square attributes.
ENDSTATE
ONSTATECURRENT
  (Details provided later.)
ENDSTATE
ONSTATEEXIT
  ; Nothing.
ENDSTATE
ENDNODE
```

NODE 14 and **ENDNODE** define the limits of the instructions. Each Node is also divided into three states: **ENTRY**, **CURRENT**, and **EXIT**.

The instructions between **ONSTATEENTRY** and the first **ENDSTATE** are executed once when the Node is entered via a **CHANGENODE** instruction. After all the **ENTRY** instructions have been executed, the Node state is automatically changed to **CURRENT**.

The instructions within the **ONSTATECURRENT** block are run repeatedly, and this state can only be left through another **CHANGENODE** instruction. This forms the bulk of Node 14, and its contents are detailed shortly.

The **ONSTATEEXIT** block, originally intended for housekeeping between Nodes, is neither used nor properly tested. However, the compiler will grumble if it can't find one.

SETALLATTRIBUTES 64 sets all bytes in the Attributes file to 64. The Attributes are where the ZX Spectrum stores colour information, in an array of 32 by 24 bytes. Each byte sets the two colours of one 8 by 8 pixel cell, and 64 is the value for black on black.

FILLSCREEN 0 sets all pixel bytes within the screen to zero, clearing all the images and messages that were shown on the title screen. The ZX Spectrum is rather slow at drawing, so hiding a dirty screen using the attributes before changing all the pixels avoids tearing.

SETLINEBANKHI 10 activates the new line bank 10, which contains the mini-game messages, in the high half of the range 0 to 127. **SETLINEBANKLO** does the same but in the low half of this range. This will make more sense when the first message is shown.

SETATTRIBUTES 0, 20, 32, 4, 68 is like **SETALLATTRIBUTES**, except the first four numbers specify which area of the screen to change. The first pair of numbers specify the XY position of the top left hand corner, and the second pair the XY dimensions of the affected rectangle. The unit of measurement for both is 8 by 8 pixel cells.

SETPGPHAREA 0, 20, 32, 4 defines where text lines will be printed in sequence. Notice that the four numbers correspond to the first four numbers after the previous instruction? It's the same area where the attributes were just changed.

SETPGPHDELAY 1, 1 defines the minimum and maximum delay, in screen frames, between text lines being drawn. The screen updates at 50hz, so one frame corresponds to 0.02 seconds. It's useful for printing a pre-defined sequence at a readable rate.

SETPGPHSPACE 0 leaves no space between text lines. **SETPGPHMODE 0** causes lines to fill from the top down, with a prompt to continue if they overflow, rather than feed from the bottom up. Individual lines can also be printed anywhere, even over this area, using the **PRINTLINE** instruction.

SETVAR 64, 0 sets the variable 64 to 0. There are 128 variables, each capable of holding an integer in the range 0 to 65535. Some more variables are initialised, then that's all for the **ENTRY** state. Here's an overview of the **CURRENT** state instructions:

```
ONSTATECURRENT
  ; Instructions.
  IFVAR==(NUM) 72, 1
    (Details provided later.)
  ENDIF
  ; Draw grid and Kenny, then start/quit prompt
  IFVAR==(NUM) 73, 1
    (Details provided later.)
  ENDIF
  ; If game is not in progress, check for start and quit keys.
  IFVAR==(NUM) 74, 0
    (Details provided later.)
  ENDIF
  ; If game is in progress, handle reveal and keys.
  IFVAR==(NUM) 74, 1
    (Details provided later.)
  ENDIF
ENDSTATE
```

The first block, as follows, prints the introduction to the game:

```
IFVAR==(NUM) 72, 1
  PRINTPGPH h0
  WAITFORKEY
  CLEARPGPH
  PRINTPGPH h2
  PRINTPGPH h3
  WAITFORKEY
  CLEARPGPH
  SETVAR 72, 0
ENDIF
```

The instructions between **IFVAR==(NUM) 72, 1** and **ENDIF** are only executed if the test condition is true. Indentation is not essential, but helps identify nested statements.

Variable 72 was set to 1 earlier, indicating that instructions are required. Because it still equals 1, the instructions are executed. The final instruction before **ENDIF** sets the same variable to 0, so that the instructions are only shown once.

PRINTPGPH h0 prints line 0 from the current high line bank, within the paragraph printing area that was defined earlier. In this case, the line bank is 10, and line 0 within the bank is “WHERE'S KENNY?**”**. **PRINTPGPH 64** would have the same effect: prefixing a number with **h** offsets it by +64, which is the space between the start of the low and high line bank.

WAITFORKEY halts execution until the player presses any key from **1** to **5**. The instruction compiler does not currently recognise any other keys.

CLEARPGPH wipes the current paragraph printing area. This appears to happen almost instantly, but with a taller area, the scrolling method used is more readily apparent.

The second block sets Kenny's position and draws a hidden play area:

```
IFVAR==(NUM) 73, 1
  ; Randomly set Kenny's position.
  GETRANDOM 3
  COPYVAR 0, 64
  GETRANDOM 3
  COPYVAR 0, 65
  ; Initialise column and row counters.
  SETVAR 76, 0 ; Column counter.
  SETVAR 77, 0 ; Row counter.
  ; Set top left printing offset.
  SETVAR 78, 5 ; X offset.
  SETVAR 79, 0 ; Y offset.
  ; Draw grid.
  CALLSUBNODE 0
  SETVAR 73, 0
  ; Show start/quit prompt.
  PRINTPGPH h5
  PRINTPGPH h6
ENDIF
```

GETRANDOM 3 sets variable 0 to a random number in the range 0 to 3. This command should only be used with the numbers 1, 3, 7, 15, 31, 63, or 127.

COPYVAR 0, 64 copies the value of variable 0 to variable 64. As noted earlier, variables 64 and 65 are used to record Kenny's XY position.

CALLSUBNODE 0 executes another block of instructions, called a Subnode, then continues from the next instruction in the Node. Subnode 0, which draws the grid and Kenny, is explained later.

Now the game is waiting for the player to press a key, either to start or quit:

```
; If game is not in progress, check for start and quit keys.
IFVAR==(NUM) 74, 0
  CALLSUBNODE 14
  IFVAR==(NUM) 22, 1
    CLEARPGPH
    ; Reset timer (start hidden.)
    SETVAR 71, 0
    COPYVAR 70, 68
    ; Start game.
    SETVAR 74, 1
  ENDIF
  IFVAR==(NUM) 22, 2
    ; Return to main title screen.
    CHANGENODE 0
  ENDIF
ENDIF
```

There are no new types of instruction, but this features the first example of a nested **IF** statement. These can currently be nested up to 64 deep.

Within the first **IF** statement, Subnode 14 checks keyboard keys **1** to **5**. It returns the value of the key pressed in variable 22, and returns a value of 0 if no key was pressed.

Two further **IF** statements check the value of the key. If the **1** key was pressed, some variables are initialised then the game is marked as in progress. If the **2** key was pressed, the game ends by returning to Node 0, the main title screen. Otherwise, nothing happens.

Finally, the most complicated block, which handles a game currently in progress:

```
; If game is in progress, handle reveal and keys.
  IFVAR==(NUM) 74, 1
    CALLSUBNODE 14
    IFVAR==(NUM) 22, 5
      (Details provided later.)
    ENDIF
    ; Is game still marked as in progress?
    IFVAR==(NUM) 74, 1
      (Details provided later.)
    ENDIF
  ENDIF
```

The first part checks if the **5** key was pressed, if Kenny was visible when the key was pressed, then runs a win or lose sequence accordingly:

```
IFVAR==(NUM) 22, 5
  ; Is reveal active?
  IFVAR==(NUM) 71, 1
    ; Is reveal at Kenny's position?
    IFVAR==(VAR) 64, 66
      IFVAR==(VAR) 65, 67
        ; Flag game as finished, set redrawing flag
        SETVAR 74, 0
        SETVAR 73, 1
        ; Show "win" message.
        PRINTPGPH h8
        PLAYSOUND 9
        PLAYSOUND 9
        PLAYSOUND 9
        PLAYSOUND 9
      ENDIF
    ENDIF
  ENDIF
  ; Is game still marked as in progress?
  IFVAR==(NUM) 74, 1
    ; Flag game as finished, set redrawing flag
    SETVAR 74, 0
    SETVAR 73, 1
    ; Show "lose" message.
    PRINTPGPH h9
    PLAYSOUND 0
  ENDIF
  ; Wait for key, then clear win/lose text and grid cell.
  WAITFORKEY
  CLEARPGPH
  SETVAR 75, 0
  CALLSUBNODE 1
ENDIF
```

PLAYSOUND 9 is the only new instruction, and this plays sound effect number 9 from the original Xenoxxx game. The ZX Spectrum has no dedicated sound hardware, so nothing else happens until the sound has finished playing.

Subnode 1 is a new addition, which will be covered shortly.

The last block handles revealing and hiding the grid squares, and has no new instructions:

```
; Is game still marked as in progress?
IFVAR==(NUM) 74, 1
  DECVAR 68
  IFVAR==(NUM) 68, 0
    IFVAR==(NUM) 71, 1
      ; Set timer and flag for inactive period.
      SETVAR 71, 0
      COPYVAR 70, 68
      ; Set reveal square attributes to black.
      SETVAR 75, 0
      CALLSUBNODE 1
    ENDIF
  ENDIF
  IFVAR==(NUM) 68, 0
    IFVAR==(NUM) 71, 0
      ; Set timer and flag for active period.
      SETVAR 71, 1
      COPYVAR 69, 68
      ; Randomly set reveal location.
      GETRANDOM 3
      COPYVAR 0, 66
      GETRANDOM 3
      COPYVAR 0, 67
      ; Set reveal square attributes to white.
      SETVAR 75, 1
      CALLSUBNODE 1
      PLAYSOUND 4
    ENDIF
  ENDIF
ENDIF
```

It's becoming hard to keep track of what all the variables do, and where nested **IF** statements start and begin. The instructions are fine for scripting sequences with no more than simple branching, but not really suitable for writing complicated games.

There are two Subnodes left to explain, and here's the first one:

```
; Draw grid, including Kenny.
SUBNODE 0
  ; Background first.
  DRAWBITMAP 50, v78, v79
  ; Then Kenny, if it's his cell.
  IFVAR==(VAR) 76, 64
    IFVAR==(VAR) 77, 65
      DRAWBITMAP 49, v78, v79
    ENDIF
  ENDIF
  ; Move relative and actual column positions.
  INCVAR 76
  ADDVAR 78, 6
  ; End of columns?
  IFVAR==(NUM) 76, 4
    ; Reset positions.
    SETVAR 76, 0
    SETVAR 78, 5
    ; Move relative and actual row positions.
    INCVAR 77
    ADDVAR 79, 5
    ; End of rows?
    IFVAR==(NUM) 77, 4
      BREAKSUB
    ENDIF
  ENDIF
  REPEATSUB
ENDSUBNODE
```

SUBNODE 0 and **ENDSUBNODE** define the limits of the instructions. Subnodes are essentially nodes with a single state, and run only once unless instructed otherwise.

DRAWBITMAP 50, v78, v79 draws the cell background picture, bitmap number 50. The second and third parameters are the XY coordinates of the top left hand corner, with the same unit of measurement as earlier commands (8 by 8 pixel cells.) However, the **v** prefix indicates that the values are found in variables, in this case variables 78 and 79.

Nearly all instruction parameters can be variable references instead of numbers. They are all covered in the **Bytecode commands** document.

INCVAR increases the value of a variable by 1. **ADDVAR 79, 5** adds the second value, which can instead be a variable reference, to the first variable (79.)

BREAKSUB and **REPEATSUB** are used for flow control, for example, handling loops with an exit condition. Subnodes are normally only executed once, but **REPEATSUB** will cause the instructions to run over and over, indefinitely. **BREAKSUB** is then the only way to return to the calling Node (in this case, when all the grid cells have been drawn.)

It's the second and final Subnode, with no new instructions:

```
; Set attributes at current reveal location.
SUBNODE 1
  ; Start at screen position for top left of grid.
  SETVAR 78, 5  ; X offset.
  SETVAR 79, 0  ; Y offset.
  ; Move screen column position by 6 * reveal's column position.
  ADDVAR 78, v66
  ADDVAR 78, v66
  ADDVAR 78, v66
  ADDVAR 78, v66
  ADDVAR 78, v66
  ADDVAR 78, v66
  ; Move screen row position by 5 * reveal's row position.
  ADDVAR 79, v67
  ADDVAR 79, v67
  ADDVAR 79, v67
  ADDVAR 79, v67
  ADDVAR 79, v67
  ; Wait for screen refresh.
  WAITONE
  ; Set grid square's attributes.
  IFVAR==(NUM) 75, 0
    SETATTRIBUTES v78, v79, 4, 4, 64 ; Bright, no flash, Black paper, Black ink.
  ENDIF
  IFVAR==(NUM) 75, 1
    SETATTRIBUTES v78, v79, 4, 4, 120 ; Bright, no flash, White paper, Black ink.
  ENDIF
ENDSUBNODE
```

If you've notice the workaround for the absence of multiplication instructions, then you're either ready to dig into Z80 assembler, or already have your own routines for that.

ZX-Modules

If you want a loading screen, and your game to run automatically after loading, then you need ZX-Modules. The two most important parts are ZX-Paintbrush, which converts images to .scr files, and ZX-Blockeditor.

The simplest way to build your own .tap file with ZX-Blockeditor is to use the official release of **XOR.tap**, but swap out the .scr file and game data block for your own. Your game data block can be extracted from .tap file produced by zDevStudio.

If you want to use full screen colour image in the program itself, then you'll need to run them through **ZX_SCR_Splitter**. This produces a monochrome .png for each image, which can then be compressed with the other bitmaps, but stores all the compressed attribute data in **ATTRS.bin**.

The attribute data is run length encoded, and the size of a whole screen. It cannot contain any attributes with the FLASH flag set, because the upper bit is instead used to indicate a run of data, i.e. the previous value repeated by the lower seven bits.

Z80 assembler

The Node instruction programming language is something like BASIC, but faster and with less features. To make full modifications, you will need to know Z80 assembler.

Assembler is computer programming at the lowest level: shuffling numbers around and comparing them. There's not many instructions to learn, but putting enough together to do something interesting is hard, because one wrong instruction can cause a spectacular crash. Tracking down more subtle bugs months after the original mistake is even worse.

So, why write ZX Spectrum programs in assembler? In short, speed and size. There's really no other way to fit everything in 48k and move graphics around at a reasonable speed. However, if you just want to make games, and don't care if they can run on a ZX Spectrum, proceed with caution.

A brief understanding of assembler, even Z80, remains useful for modern computer programming because fundamental concepts have not changed. Writing whole programs in assembly is masochistic if your computer has gigabytes of memory, and has compilers that can turn high-level languages into something almost as fast.

If you're serious about learning Z80 assembler, then there are plenty of tutorials available online, and also scans of old ZX Spectrum programming books. I still occasionally refer to my copy of *Spectrum +2 Machine Language For The Absolute Beginner*.

Memory map

Even if you're not planning to write any assembler, the **Memory map** within the **Docs** folder will show you how much free space there is to work with. It has been laid out with gaps that allow a little more of everything: assembler, Node data, graphics, and text.

Now for the final programming section, which assumes some familiarity with hexadecimal and assembler.

XOR from top to bottom

The loading routine starts at $5F00, because the earlier space is required for the BASIC program which pokes the loading routine into memory. The BASIC is no longer required after loading, so it is used for a buffer and some stacks instead.

The attribute buffer is handy for making complex changes without tearing, and is also used for special effects like the fake reset anomaly. The two stacks are used to handle nested statements and Subnode calls.

The loading and initialisation routine only run once, so can be overwritten later if you need another small scratch space. The screen, interrupts, and stack are initialised, then control is handed to the main loop, in fast memory. The rest of the slow memory is used mainly for text data, because reads are small and not speed-critical.

Sound generation is fairly crude, generating only one tone channel. However, the apparent volume can be reduced by using a relatively short high or low period pulse, rather than defining an equal square wave.

XOR-PARSER is the first big block of code, and it interpret the bytecode generated by the Node compiler. It's essentially a look-up table, with some special cases for flow control instructions. There are plenty of gaps if you want to add your own instructions, though you'll need to modify the Python compiler accordingly.

XOR-UTILITIES contains generic routines for dealing with index tables, screen addressing, etc.

XOR-DRAWING decompresses bitmap data. All operations use the 32 by 24 attribute grid coordinate system; there are no “per pixel” routines. This routine only handles compressed bitmaps and draws directly to the screen, so clipping is not implemented.

The bitmap compression software does an exhaustive search of all previous compressed image data for any blocks of eight bytes that match the cell currently under consideration. Because a successful match only stores a memory pointer, the routine could be modified to also search other data for matches: compressed text, machine code, even the ROM.

XOR-PRINTING handles decompressing text lines, and also drawing them within a defined rectangle, automatically wrapping and scrolling as required.

The text compression software could be optimised to consider substitution of any character sequence, in addition to whole words, when building the dictionary table.

XOR-SPECIFIC contains routines that are probably not much use outside of the Xenoxxx Funtime Show. The corresponding bytecode instructions are confined to the range $E0 to $FF, so this is the best block to replace if making your own game from scratch.

The rest of **XOR** contains routines that were neither sufficiently generic nor specific, and imports data files. It also includes two embedded data blocks: the character bitmaps, and the tunnel map.

The character bitmaps were made with some ancient Java software I should probably update to Python. The tunnel map is just the CSV export of **Drakelow ZX,** with a prefix on each line. It could possibly be stored as a compressed attribute file, because it has exactly the same dimensions, and the upper bits are only used to indicate inactive cells.

The future

I am now making a new game building on the same tools, so plan to filter back any improvements into a further revision of the Xenoxxx Funtime Show.

Until then, please send any feedback or questions to: *comments at arbitraryfiles.com.*

Revised 21-12-2017.
Uploaded 22-10.2017.