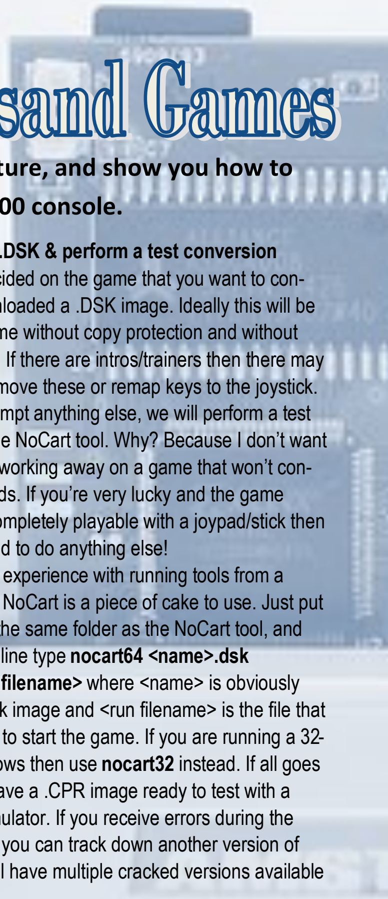


The Console of a Thousand Games

We pick up where we left off last time from our C4CPC feature, and show you how to convert your own titles so that they're playable on a GX4000 console.

As promised all the way back in issue 1, we'll be taking you through the process of converting a game to the .CPR format for use with the C4CPC device. A huge amount of CPC games have already been converted, but of course there are still some exceptions among the thousands of titles released for the CPC and it's quite possible that one of your favourite games hasn't made it yet. Also, I imagine that there are some of you who are just curious about the process.

To find the full list of what has been converted so far, check the 'Converted GX4000 Software' page on the CPCWiki (of course) at: **http://www.cpcwiki.eu/index.php/Converted_GX4000_Software**

If you've found that one of your favourite titles is not on that list and you're dying to get it up and running on your GX4000, then read on…

For this process you will need:

- A .DSK image of the game – preferably without copy protection (i.e. a cracked version)
- An emulator with debugging functionality such as WinAPE, that also supports .CPR images for you to perform an initial test of the game (we'll be using WinAPE in this guide, but the principles should apply to other emulators).
- A HEX/sector editor tool – e.g. HxD or CPCDiskXP.
- NoCart application by Kevin Thacker
- A basic knowledge of Hexadecimal and memory addresses.
- A C4CPC if you want to play it on a GX4000

Step One - Find a .DSK & perform a test conversion

So, you've decided on the game that you want to convert and have downloaded a .DSK image. Ideally this will be a version of the game without copy protection and without using compression. If there are intros/trainers then there may be extra work to remove these or remap keys to the joystick.

Before we attempt anything else, we will perform a test conversion using the NoCart tool. Why? Because I don't want you to spend ages working away on a game that won't convert or run afterwards. If you're very lucky and the game you've chosen is completely playable with a joypad/stick then you won't even need to do anything else!

If you have any experience with running tools from a command line then NoCart is a piece of cake to use. Just put the .DSK image in the same folder as the NoCart tool, and from the command line type **nocart64 <name>.dsk <name>.cpr <run filename>** where <name> is obviously the name of the disk image and <run filename> is the file that you run on the disk to start the game. If you are running a 32-bit version of Windows then use **nocart32** instead. If all goes well then you will have a .CPR image ready to test with a C4CPC or in an emulator. If you receive errors during the process then see if you can track down another version of the game, many will have multiple cracked versions available online.

Step Two - Play and study the game

The next step is to try running your new .CPR image in either a real GX4000/Plus via a C4CPC or in an emulator (with only 64KB and Plus features enabled). Even if you know that the game you're trying is not 128KB only, some cracked versions need 128KB even if the original ran in

	BIT							
Line	7	6	5	4	3	2	1	0
&40	F .	ENTER	F3	F6	F9	Cursor ↓	Cursor →	Cursor ↑
&41	F0	F2	F1	F5	F8	F7	COPY	Cursor ←
&42	CONTROL	\	SHIFT	F4	]	RETURN	[	CLR
&43	.	/	:	;	P	@	-	^
&44	,	M	K	L	I	O	9	0
&45	SPACE	N	J	H	Y	U	7	8
&46	V	B	F / Joy2 Fire2	G / Joy2 Fire1	T / Joy2 →	R / Joy2 ←	5 / Joy2 ↓	6 / Joy2 ↑
&47	X	C	D	S	W	E	3	4
&48	Z	CAPSLOCK	A	TAB	Q	ESC	2	1
&49	DEL	Joy1 Fire 3	Joy1 Fire 2	Joy1 Fire 1	Joy1 →	Joy1 ←	Joy1 ↓	Joy1 ↑

64KB. If the game does not run, then once again, try to see if you can find another version.

As a real GX4000 doesn't have a keyboard it is important to check to see if the game is playable from joystick alone; many games require key presses to get through the menus or enter a high score even if the game itself is playable with only a joystick.

Make a note of any keypresses that you are required to make to get into and play the game. If you find that a keypress is required on the title screen to switch to joystick or start the game then make a note of this along with the keypresses for other things like 'redefine keys'. In the game itself make a note of the default keyboard controls and also the keys for things like pause and quit.

Step Three - Re-map any required keyboard inputs

For us to map the keyboard input to the joystick, it is useful to understand a little bit about how keyboard input is read on the CPC. As far as the CPC is concerned, each of the possible joystick inputs (Joy Up, Joy Down, Joy Left , Joy Right, Joy Fire 1 and Joy Fire 2) are handled as if they were just another key on the keyboard. If you look at the table (opposite-bottom) then you'll see that there are a total of 80 keys that can be recognised; 10 lines labelled &40 to &49 with 8 bits on each labelled bits 7 down to 0. So what we want to do is change the expected keyboard input to one of the available buttons on the joypad. If the game requires the player to press '1' to start (such as in Potsworth & Co, which we'll be using as an example) which is bit 0 of line &48, we would want to map this to bit 4 of line &49 which is the primary fire button of Joystick 1.

Bizarrely, it is actually the CPC's sound chip (AY-3-8912) that is responsible for scanning the keyboard. The AY chip is accessed via a chip known as the 8255 PPI. This is an interface chip that lets the CPC's Z80 CPU communicate with different elements of the CPC's hardware, and that is done via three 'ports' labelled A-C (F4, F5, F6) and a control register (F7) with values sent to these using the **OUT** command. When this command is used, register **B** will contain the value of port/control register (i.e. F6) and another register will contain the value to be sent (e.g. **ld bc,#f782** followed by **out(c),c** will send the value #82 to the control register #F7).

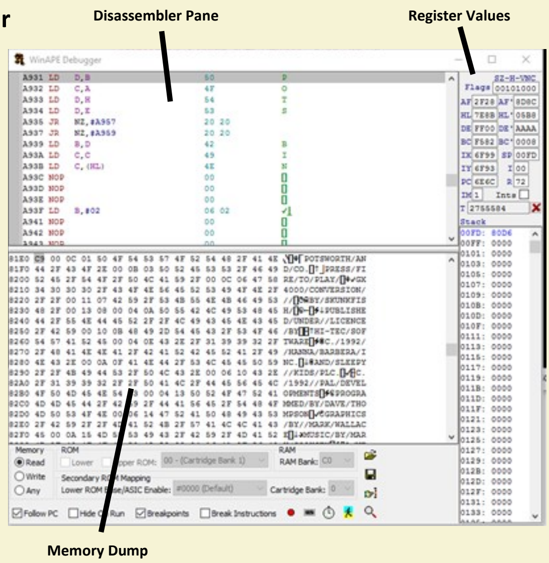
To read the keyboard, the code will set the PPI for Output on ports A & C (#82 to #F7) as we're sending data to the PSG, then it will select register 14 of the PSG (the IO register used for reading keyboard inputs) by writing 14 to Port A (#0E to #F4) followed by #C0 to Port C (#C0 to #F6) before writing 0 to Port C (#00 to #F6) to set the PSG to inactive (this prevents it going funny when switching from output to input). Next, Port A of the PPI is set to input by sending #92 to #F7 (the control register). The value of the keyboard line to read (e.g. #40) is then sent to Port C (#F6) before finally the keyboard value is read from port A into a register, this time using the **in** command.

Does that make an ounce of sense? We only have space here to go briefly into how the CPC reads the keyboard input, so we suggest that you swot up with some

Feel at home with the WinAPE Debugger

Even if you've never used the Debugger functionality of WinAPE before, it won't take long to familiarise yourself with it. Pressing **F7** at any time will pause the emulation and allow you to view a disassembly of the currently running program and examine the contents of the memory or registers. Pressing **F7** further times will execute the code, one instruction at a time. Breakpoints can be set, either at a set point in memory (by highlighting the address and pressing **F5**) or when an I/O port is accessed which mean that the emulator will pause at that point, opening the debugger automatically for you.

There are a couple of other useful key combinations to remember. **Ctrl+G** will allow to enter a memory address to jump to in either the Disassembler Pane or the Memory Dump Pane. **Ctrl+F** allows you to search for Text, Hex values or Assembler commands. Pressing **F5** will set a breakpoint at the currently selected line in the Disassembler Pane.



```
A931 LD   D,B          50      P
A932 LD   C,A          4F      O
A933 LD   D,H          54      T
A934 LD   D,E          53      S
A935 JR   NZ,#A957     20 20
A937 JR   NZ,#A959     20 20
A939 LD   B,D          42      B
A93A LD   C,C          49      I
A93B LD   C,(HL)       4E      N
A93C NOP               00
A93D NOP               00
A93E NOP               00
A93F LD   B,#02        06 02
A941 NOP               00
A942 NOP               00
```

```
81E0 C9 00 0C 01 50 4F 54 53 57 4F 52 54 48 2F 41 4E  [illegible]POTSWORTH/AN
81F0 44 2F 43 4F 2E 00 0B 03 50 52 45 53 53 2F 46 49  D/CO.[illegible]PRESS/FI
8200 52 45 2F 54 4F 2F 50 4C 41 59 2F 00 0C 06 47 58  RE/TO/PLAY/[illegible]GX
8210 34 30 30 30 2F 43 4F 4E 56 45 52 53 49 4F 4E 2F  4000/CONVERSION/
8220 2F 2F 00 11 07 42 59 2F 53 4B 55 4E 4B 46 49 53  //[illegible]BY/SKUNKFIS
8230 48 2F 00 13 08 00 04 0A 50 55 42 4C 49 53 48 45  H/[illegible]PUBLISHE
8240 44 2F 55 4E 44 45 52 2F 2F 4C 49 43 45 4E 43 45  D/UNDER//LICENCE
8250 2F 42 59 00 10 0B 48 49 2D 54 45 43 2F 53 4F 46  /BY[illegible]HI-TEC/SOF
8260 54 57 41 52 45 00 04 0E 43 2E 2F 31 39 39 32 2F  TWARE[illegible]C./1992/
8270 2F 48 41 4E 4E 41 2F 42 41 52 42 45 52 41 2F 49  /HANNA/BARBERA/I
8280 4E 43 2E 00 0A 0F 41 4E 44 2F 53 4C 45 45 50 59  NC.[illegible]AND/SLEEPY
8290 2F 2F 4B 49 44 53 2F 50 4C 43 2E 00 06 10 43 2E  //KIDS/PLC.[illegible]C.
82A0 2F 31 39 39 32 2F 2F 50 41 4C 2F 44 45 56 45 4C  /1992//PAL/DEVEL
82B0 4F 50 4D 45 4E 54 53 00 04 13 50 52 4F 47 52 41  OPMENTS[illegible]PROGRA
82C0 4D 4D 45 44 2F 42 59 2F 44 41 56 45 2F 54 48 4F  MMED/BY/DAVE/THO
82D0 4D 50 53 4F 4E 00 06 14 47 52 41 50 48 49 43 53  MPSON[illegible]GRAPHICS
82E0 2F 42 59 2F 2F 4D 41 52 4B 2F 57 41 4C 4C 41 43  /BY//MARK/WALLAC
82F0 45 00 0A 15 4D 55 53 49 43 2F 42 59 2F 4D 41 52  E[illegible]MUSIC/BY/MAR
```

```
SZ-H-VNC
Flags 00101000
AF 2F28  AF' 8D8C
HL 7E8B  HL' 06B8
DE FF00  DE' AAAA
BC F582  BC' 0008
IX 6F99  SP 00FD
IY 6F93  I  00
PC 6E6C  R  72
IM 1     Ints
T 2755584
Stack
00FD: 80D6
00FF: 0000
0101: 0000
0103: 0000
0105: 0000
0107: 0000
0109: 0000
010B: 0000
010D: 0000
010F: 0000
0111: 0000
0113: 0000
0115: 0000
0117: 0000
0119: 0000
011B: 0000
011D: 0000
011F: 0000
0121: 0000
0123: 0000
0125: 0000
0127: 0000
0129: 0000
012B: 0000
012D: 0000
012F: 0000
0131: 0000
0133: 0000
```

Memory: (●) Read ( ) Write ( ) Any
ROM: ☐ Lower ☐ Upper ROM: 00 - (Cartridge Bank 1)
RAM Bank: C0
Secondary ROM Mapping: Lower ROM Base/ASIC Enable: #0000 (Default) Cartridge Bank: 0
☑ Follow PC ☐ Hide On Run ☑ Breakpoints ☐ Break Instructions

further reading if you're keen on attempting this yourself (see 'further reading' at the end of the article).

Unfortunately, there's no one way that the keyboard scanning code will appear, although you may learn to spot it after a while. Luckily, WinAPE can set a breakpoint on I/O reads so you don't have to understand it too much.

In WinAPE, whilst on the title screen of the game, we will press **F7** game to enter the debugger and click the small Red circle at the bottom of the window to bring up the Breakpoints window. Click the 'Input/Output' tab and click 'Add', then select 'Keyboard Read' under type and click OK. Close the debugger window and it will start the emulation again. The emulator will stop when it detects that keyboard input is being read and bring the debugger back up. You'll need to clear the breakpoint to be able to continue as normal.

Once in the debugger you can press (or hold) **F7** to cycle through the code a single instruction at a time. The title screen is a great place to see the keyboard scanning in action because quite often that's pretty much the ONLY thing happening on the title screen.

If it's reading multiple lines then it will likely store the current key status in an area of memory, ready to be read by the game code to determine what to do next.

This is what both Potsworth & Co and Scooby Doo & Scrappy Doo do (don't step in it). Both titles were coded by the same author so work in a very similar way. In the code below that we found by setting the breakpoint in Potsworth, it sets the start location for the keymap in memory at **#FFF6**, reads the first keyboard line (**&40**) into that memory address, increases the above values to **#FFF7** and **&41** and continues in a loop reading each line of the keyboard until the memory address attempts to go past **#FFFF** (to 00), at which point it exits the loop.

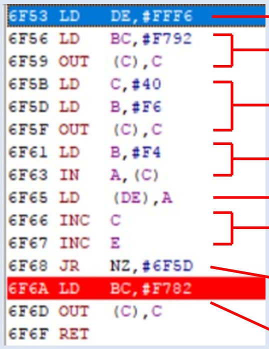
So now that we know how the game reads the input from the keyboard, we need to change how it behaves when it receives that input. As per the keyboard matrix table, by checking which bits are set for each line the CPC can determine which keys are currently being pressed. We know that on the title screen of Potsworth it expects the user to press '1' to start the game (bit 0 of line &48), which we want to change to 'FIRE1' on Joystick/Joypad 1. 'We know that the game is storing inputs from line &48 in #FFFE, so we can search for **LD A,(#FFFE)** in the debugger (Select disassembler window, Ctrl+F and then search for that under the 'Assembler' tab) to see where it attempts to use that value. In your game it might be worth searching for the **BIT** instructions instead, particularly if you know which keys it is looking for.

We soon find that instruction at address **#80DC** (below-right) which is followed by **BIT 0, A** and **JP Z,#4F00**. This looks exactly what we're after, but we set a quick breakpoint to confirm that this code is executed on the title screen (it is). So what is it doing? After loading the value of the keyboard line &48 from **#FFFE** the code is checking if **bit 0** is set and if so jumps to **#4F00** (to start the game).

So how do we set this to use FIRE 1? We simply need to amend the line and bit that it is reading to **#FFFF** (line &49) and **bit 4** (Joy 1 FIRE1). Just be aware that this will also impact the next part of the code, as where it previously looked for bit 1 of line &48 ('2' on the keyboard) to redefine the keys, this will now be looking at be looking at line &48 as well.

When making changes, take before and after screenshots of each part of the code that you amend so that you have a record of what has been changed.

To make the amendment in WinAPE, you will need to jump to the instruction in the Memory Dump pane (click in the pane, press **Ctrl+G** and enter **80DC** jump to that part of the code. We want to amend **3A FE FF** to read **3A FF FF** so we will click in the location that needs changing and just enter the change. The Hex values that you see are translation of the **ld a,(#FFFE)** command. Each Z80 instruction is represented by a Hex value known as an 'Opcode' which is followed by an operator if required, such as the memory address. In this case, the Opcode for **ld a,(**)** is **3A** and the memory address is **FF EE.** You'll noticed that the bytes in the address are reversed (**FE FF** rather than **FF FF).** This is because the Z80 follows a 'little-endian' format, where it



```
6F53 LD    DE,#FFF6
6F56 LD    BC,#F792
6F59 OUT   (C),C
6F5B LD    C,#40
6F5D LD    B,#F6
6F5F OUT   (C),C
6F61 LD    B,#F4
6F63 IN    A,(C)
6F65 LD    (DE),A
6F66 INC   C
6F67 INC   E
6F68 JR    NZ,#6F5D
6F6A LD    BC,#F782
6F6D OUT   (C),C
6F6F RET
```

-Set first address to read into

-Tell PPI that Port A is input & Port C is output.

-Send keyboard line value to Port C

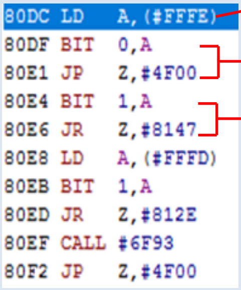
-Read data from PPI port A

- Store value in current keymap address

- Increase keymap address and keyboard line

- Loop until end of keymap area is reached

- Return PPI port A to output mode



```
80DC LD    A,(#FFFE)
80DF BIT   0,A
80E1 JP    Z,#4F00
80E4 BIT   1,A
80E6 JR    Z,#8147
80E8 LD    A,(#FFFD)
80EB BIT   1,A
80ED JR    Z,#812E
80EF CALL  #6F93
80F2 JP    Z,#4F00
```

- Load the value of #FFFE (line &48) into the accumulator

- Check to see if bit 0 of that line is set ('1' on the keyboard), if so jump to #4F00.

- Check to see if bit 1 of the same line is set ('2' on the keyboard), if so jump to #8147

stores the least significant byte first. Since you may need to amend some Opcodes (particularly when amending the **bit** command below), we advise having a browser tab open at **http://clrhome.org/table/** for reference; this site lists all of the available Z80 Opcodes in a handy table format. As soon as the change is made you should see the disassembly window update with the new instruction.

The Opcode for **bit 0, a** is **CB 47** but we want to read the value of **bit 4** instead. A quick glance at the reference table tells us that the required Opcode is **CB 67** so we make the amendment in the same way as above. You'll then want to test the change, but before you do we recommend that you save a snapshot (**F6)** so that you can get back to the title screen easily. With that done, trying the Fire button reveals whether the change has worked or not (it has, yay!). Using the same method, you'll want to move any other inputs on the title screen to the Joypad as required.

With everything working as it should, we attempt to go into the game to redefine the main controls. As we did for the title screen, set an I/O breakpoint on Keyboard read. As it turns out, the game uses the same code as the title screen (very efficient) storing a keymap in the same location (**#FFF7-#FFFF).** The default keys are Q, A, O, P, so we start by trying to find where it checks for the 'Q' key (Up). Back to the table on page 10, we know that Q is on line **&48** which is stored in **#FFFE** so we will search in the disassembler for the command **ld a,(#FFFE).** This takes us to #6F7B, but what we find here rather than a **BIT** command following the **LD A,(#FFFE)** there is **AND #08.**

This is another way of checking which bits are set, by comparing a number against the accumulator.

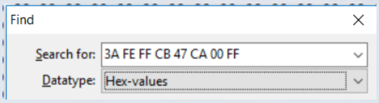
Seeing **AND #08,** it may not be immediately obvious which key is checking for, but if you were to convert the number to binary it would shed some light. It becomes **AND %00001000 -** i.e. it's checking if Bit 3 has been set on line &48, which looking back at the table is the **Q** key, representing Up in the game. We want to change this to Joystick/ Joypad Up which is line &49, bit 0 so we need to change **LD A,(#FFFE)** to **LD A,(#FFFF)** and **AND #08** to **AND #01** (representing **%00000001**, so the 0 bit is set) in the same way as we did before, taking a screenshot before we make the change. Carry on and do this for each of the controls, saving a screenshot of the changes and a snapshot when done. Start the game again and test out your new controls. Is something not quite working as expected? Reload the snapshot, go back to your changes and see if something needs amending.

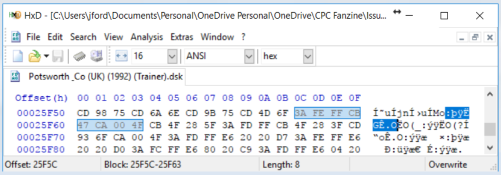
If it does appear to be working, play the game thoroughly and confirm that you can indeed play the game properly with just a joypad/stick. Remember things like high-score tables when you die or level code entry. If there is one of these can the letters be inputted without a keyboard? If not, you'll need to go through the process again and work out a way so they can (or at least to skip past the entry).

If you're not having any luck finding the keyboard routines in a game, it may be that it is using the firmware routines for this instead. **Call &BB1E** (KM TEST KEY) checks to see if a particular key has been pressed, with the accumulator (A) containing the key reference, The key references it uses are in the user manual, or can be calculated from our key matrix table. For example, **Q** is key 67, which can be calculated from our table by taking the last digit of the line number (8), multiplying that by 8 (giving 64) and finally adding the bit (3), giving a grand total of 67.

Step Four - Make the changes with a disc/file editor

So, if you paid attention in Step Three you should not have some screenshots of the code before and after your changes (well, it's actually the hex values just to the right that we're after). Rather than attempting to save a new binary file with our changes, we are going to edit the .DSK file directly. There are different ways that this can be done, but we tend to use a tool called **HxD** which is available for free from **http://www.mh-nexus.de.** Using this tool, you will need to search for the pre-modified hex values from your screenshots and apply the same modifications to them. After opening the .DSK image in HxD, press Ctrl+F to search for the series of hex values (make sure to change the datatype to 'Hex Values'). The more values you enter, the greater the chance of you finding the right area of code.




HxD - [C:\Users\jford\Documents\Personal\OneDrive Personal\OneDrive\CPC Fanzine\Issu...

File Edit Search View Analysis Extras Window ?
16 ANSI hex
Potsworth _Co (UK) (1992) (Trainer).dsk

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00025F50  CD 98 75 CD 6A 6E CD 9B 75 CD 4D 6F 3A FE FF CB   Í˜uÍjnÍ›uÍMo:þÿË
00025F60  47 CA 00 4F CB 4F 28 5F 3A FD FF CB 4F 28 3F CD   GÊ.OËO(_:ýÿËO(?Í
00025F70  93 6F CA 00 4F 3A FD FF E6 20 20 D7 3A FE FF E6   “oÊ.O:ýÿæ  ×:þÿæ
00025F80  20 20 D0 3A FC FF E6 80 20 C9 3A FD FF E6 04 20     Ð:üÿæ€ É:ýÿæ.
```

Offset: 25F5C   Block: 25F5C-25F63   Length: 8   Overwrite

When you've found the area to change, overwrite the existing values with your new ones. Once complete, save the .DSK with a new filename and check it with your emulator to make sure it works as expected. All that's left now is to convert the DSK image again as we did in Step 2 and copy the game to your C4CPC. Congratulations, you've just expanded the games library of the GX4000!

Further reading:

http://www.wacci.org.uk/magazine/138/138_06.html
http://www.cpcwiki.eu/index.php/Programming:Keyboard_scanning
http://lronaldo.github.io/cpctelera/files/keyboard/keyboard-txt.html