sinclair

# ZX Spectrum Next

## User Manual

Written and Illustrated by
Phoebus R. Dokos, BSc (Hons)

*With extracts from:*
ZX Spectrum +3 User manual
by **Ivor Spital**, **Cliff Lawson** and **Rupert Goodwins**
ZX Spectrum BASIC programming manual
by **Steven Vickers** and **Robin Bradbeer**

Edited by:
Mike Cadwallader, Uwe Geiken
Darren Grayson, Matt Langley
David Saphier, Paulo Silva
Julian Smith and Steve Smith

With invaluable contributions by:
Alvin Albrecht, Garry Lancaster, Simon N Goodwin,
Simon Brattel and Kev Brady

Copyright © 2020-2024 Phoebus Dokos / SpecNext Ltd – London, United Kingdom

This work is licensed under a CC BY-NC-SA 4.0 International License.
http://creativecommons.org/licenses/by-nc-sa/4.0/

Cover Illustration: **Jonathan M Betts** (www.artstation.com/jonathanmbettsart)
Cover Layout: **Phoebus R Dokos** (www.dokos-gr.net)

Listings set in:
ZX Spectrum Next Mono typeface
Copyright © 2017–2024 by Phoebus R Dokos

THIRD EDITION

ISBN: 978-1-5272-5496-1

*To Georgia*

Copyrights / Trademarks

Sinclair and ZX Spectrum are copyright © Amstrad/Sky plc and are used under license
Spectrum Next and System/Next are copyright © SpecNext Ltd
The NextCore is © Alvin Albrecht
+3e, ResiDOS, IDEDOS, NextZXOS and NextBASIC are copyright © Garry Lancaster
TBBLUE is © Victor Trucco and Fabio Belavenuto
Zeus is © Neil Mottershead and Simon Brattel
NextPi, NextPi2, SPUI and QE are © D. Rimron
ZX-UNO is © The ZX-UNO Team (Superfo, Avillena, McLeod, Quest, Hark0)
divMMC is © Mario Prato
CP/M is © Lineo Inc.
esxDOS is © Miguel Guerreiro / Papaya Dezign
The ZX80/ZX81 emulators are © Paul Farrow
Gosh Wonderful and Looking Glass are © Geoff Wearmouth
nxtp and NxTel are © Robin Verhagen-Guest
NextGuide, ED and Odin are © Matt Davies
SpectraMon and ZIP/Next are © Simon N Goodwin
ZXDB-dl and GetIt are © David Saphier
vDrive$^{zx}$ is © Charlie Ingley
ULAplus is ™ ZX Design and Media
All other names and trademarks used herein are ©/™ of their respective authors/owners

# Chapter 1 – Basic Programming Concepts

## Introduction

If you read through the *Quick Start Guide*, included with your new ZX Spectrum Next, you've already had a brief introduction of the screen, keys, editing and *NextBASIC* in general which means you're ready to start programming your computer! If not, you can either go along and you'll figure things you've missed along the way – or – go back and have a quick read of the *A (Next)BASIC Primer* section! Either way, you need to reset your ZX Spectrum Next, go to the *Startup Menu* and select *NextBASIC*. Press **ENTER** and we're ready to start!

## PRINT, LET, programs and line numbers

Type in the following two lines:

```
20 PRINT a
10 LET a=10
```

so that the screen looks like this:

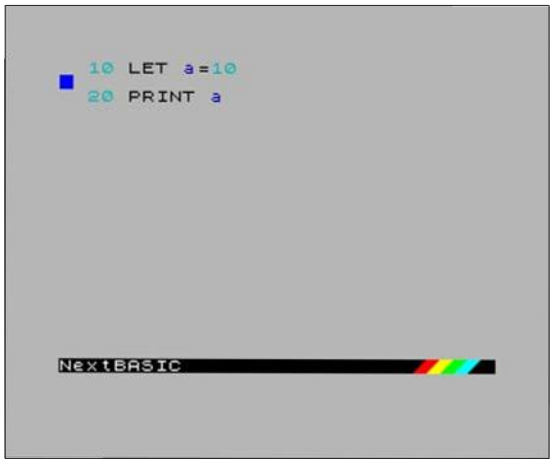



*Fig. 1 – Entering program lines in NextBASIC*

First of all congratulations! You just wrote a *computer program* which stores a number in the computer's memory, later recalls it and displays it. Let's see for a moment exactly how you've done that:

- Since these lines began with numbers (as you already know from the Quick Start Guide), they were not obeyed immediately but stored as *program lines*. You will also have noticed here that the respective line numbers govern the order of the lines within the program: the lower the number the earlier (higher in the list) it appears. This matters most when the program is run, but it also governs the order of the lines that you see on the screen now.
- By using the command **LET** you've instructed the computer to await an *assignment* – the assignment itself is indicated by the = (equals sign). *Assignment* is the pairing of the numeric value **10** to a *variable* named **a**.

Let's enhance our program a bit more. Type:

```
15 b=15
```

and press **ENTER**. Line 15 gets inserted between lines 10 and 20 and the screen is reformatted. If the lines' numbers had only an interval of **1**; if for example they had been numbered 1 and 2 instead of 10 and 20 it would have been impossible to insert another line in-between. Line numbers must be whole numbers between **1** and **9999**, and that is why,

when first typing-in a program, it is good practice to leave large enough intervals in-between the numbers.

You'll also notice, that for line 15 there's no **LET** keyword although the = remains. That is because **LET** is optional and it's implied from the *assignment* alone. Functionally therefore, lines 10 and 15 are identical. For this chapter, we will keep using **LET** so you can see the assignments clearly but further on, we will skip them altogether as they make for much more readable code.

Note here that we will do the same with the printed representation of *Syntax Highlighting* as it requires a contrast with the background which a printed manual doesn't provide.

## Variables and Arrays

Before we continue further, let's take a pause and discuss what the letters **a** and **b** in the examples above are called. We call these *variables* because they represent locations in the computer's memory where we can temporarily store information to be recalled and used at any time a program is being executed. There are two types of variables by usage: *Global* and *Local*. Global variables apply to an entire *NextBASIC* program and these are the ones we will talk about here. Local variables, apply only to subprogram areas we call *procedures* and *functions* and they will be discussed in the respective chapters.

*NextBASIC* can store two types of information in memory: *numbers* and *text*. Numbers are further separated into *floating point* and *integers*. Text variables are called *strings* and they will be discussed in *Chapter 7.* Furthermore, *NextBASIC* can group together variables of the same type and refer to them collectively. These groupings are called *arrays*.

There are some restrictions in the naming and quantity of available variables and arrays as you can see in the following table according to their type. It is advisable to make use of integer variables over their regular numeric counterparts despite their restrictions[1] at least where speed of execution is concerned.

| | Integer variables | Numeric variables | String Variables |
|---|---|---|---|
| Qty | Fixed 26 | Limited only by memory | Limited only by memory |
| Naming | Single character prefixed by the % symbol | Combination of characters and numbers | Combination of characters and numbers suffixed by the $ symbol |
| Arrays | Fixed 26 with maximum 64 elements (0...63)<br>Extensible size and dimensions (by reducing the number of available arrays) | Limited only by memory<br>(Indices are *1-based*) | Limited only by memory<br>Indices are *1-based*) |

*Table 1 – Types of NextBasic variables*

## Assignments

We saw earlier that using **LET** with a variable together with the symbol = and some value is called an assignment. What actually happens is that **LET** instructs *NextBASIC* to move a value (numeric or character) into a location in the computer's memory which we can later identify and recall by an easy-to-use name (*See Table 1 above*). Unlike previous versions of Sinclair BASICs that *required* the **LET** command and only allowed a single **LET** command per assignment, *NextBASIC* allows the omission of the **LET** keyword altogether (as the assignment operator = implies its use anyway) while, at the same time, multiple assignments of variables per **LET** command. In other words, the full form of **LET** is:

[**LET**] *variable1 [,[variable2....variablen]] = value1[,[value2...valuen]]*

Moreover, assignments allow multiple destinations and a mix of types of value and variables. Some examples of the above are:

```
LET x,y=10,20
```

1 *Integer variables in NextBASIC are 16-bit (unsigned or signed). That means that they accept values from 0 to 65535 (or from -32768 to 32767)*

```
x,y=10,20
```

which are equivalent, or in a more descriptive manner:

```
LET numberA,numberB,stringA$,stringB$ =
1,2,"hello","goodbye"
```

One, extremely handy functionality of assignments is that in the case of assignment of multiple variables, if there are fewer values after the = operator than the variables before it, then all the remaining variables get initialised to that specific value. The following line:

```
a,b,c,d,e,f=0
```

will create variables **a**, **b**, **c**, **d**, **e** and **f** and set them all to **0**. Similarly the following line:

```
a,b,c,d,e,f=10,20,30,1
```

will assign **10** to variable **a**, **20** to variable **b**, **30** to variable **c** and **1** to variables **d**, **e** and **f**. In the example above we could remove altogether line 10 and instead enter:

```
15 a,b=10,15
```

which is functionally equivalent to both lines 10 and 15!

Finally, assignments can be made cumulative with the use of special combination operators as seen in the table below:

| Assignment Operator | Numeric Variables | String Variables |
|---|---|---|
| += | Increases variable by the value assigned | Concatenates string variable with the string assigned |
| -= | Decreases variable by the value assigned | – Not Applicable – |
| *= | Multiplies variable by the value assigned | Replicates the string variable as many times as the numeric value assigned |
| /= | Divides variable by the value assigned | – Not Applicable – |
| ^= | Raises variable to the power assigned | – Not Applicable – |
| &= | ANDs variable with the value assigned | – Not Applicable – |
| \|= | ORs variable with the value assigned | – Not Applicable – |
| ^\|= | XORs variable with the value assigned | – Not Applicable – |
| <<= | Shifts left the variable as many positions as the value assigned | – Not Applicable – |
| >>= | Shifts right the variable as many positions as the value assigned | – Not Applicable – |
| MOD= | Performs a MODulo operation on the variable with the value assigned | – Not Applicable – |

*Table 2 – Accumulation assignments*

This allows us to be a bit more terse when writing a program by reducing the amount of text we have to type, making for some more readable code. Consider the following example:

```
10 LET a=10
15 LET b=15
20 LET c=a
30 LET a=a+b
40 LET b=b+c
```

Using the information we've just learned, we can rewrite it to use multiple assignment statements as well as cumulative assignment operators like so:

```
15  a,b=10,15
20  a,b+=b,a
```

It's obvious from the example above that after skipping both the **LET** keyword and the long form of assignment, our program suddenly became more readable and much easier to write!

## Labels

Apart from the line numbers (which we will see how to refer – and jump to – in the following sections) sometimes we need a way to identify and/or jump to a selected *NextBASIC* statement, maybe even within a multi-statement line. For this reason, *NextBASIC* provides us with a facility called *labels*. Labels are identified by the **at** (@) symbol prefix and a name following the naming guidelines for a *procedure* (See *Chapter 4*) and they are defined within a program starting with either the line number or the **colon** *statement separator* ( **:** ) if they're not defined at the start of a line. Their definition can appear anywhere within a program:

```
20  @onelabel: PRINT a+b
30  PRINT a+b: @anotherlabel
```

Labels can be used in lieu of line numbers with the following keywords: **GO TO**, **BANK...GO TO**, **GOSUB**, **BANK...GOSUB**, **LIST**, **BANK...LIST**, **SAVE...LINE**[2] and **EXIT**. See relevant section for each keyword's proper syntax. **BANK** commands are all discussed in length in *Chapter 23 – The Memory*

## Using LIST, RUN and cursors to edit and run programs

Going back to our program, you will need to change line 20 to:

```
20  PRINT a+b
```

You could type out the replacement in full, but it is easier to move the cursor (using the cursor keys) to just after the **a**, and then type:

```
+b  (without ENTER)
```

The line at the bottom should now read:

```
20  PRINT a+b
```

Press **ENTER** and it will replace the old line 20, so that the screen looks like this:

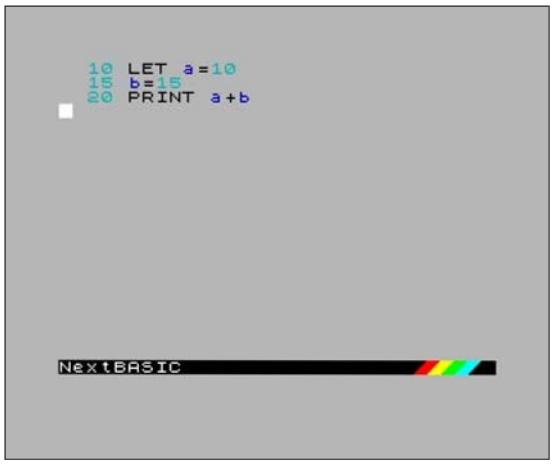



*Fig. 2 – Editing a program*

2 *In the case of SAVE...LINE@label, the saved program will autostart from the beginning of the line that contains the label even if it's not the first statement in the line* [2]

Run this program using **RUN** and **ENTER** and the sum will be displayed (**25**). Run the program again and then type:

```
PRINT a, b
```

The variables are still there, even though the program has finished. If you enter a line by mistake, say:

```
12 b=8
```

it will go up into the program and you will realise your mistake. To delete this unnecessary line, type:

```
12
```
(with **ENTER** of course)

Line 12 will disappear, and the cursor will appear where line 12 used to be.

Now type:

```
30
```
(and **ENTER**)

This time, the program cursor will appear after the end of the program (having tried to find line 30 and failed). If you enter any line number that does not exist, *NextBASIC*'s *Editor* will place the cursor where it thinks the line would have been if it existed. This can be a useful way of moving around large programs, but beware – it can be very dangerous because if the line really did exist before you entered the number, it wouldn't exist afterwards (refer to the line 12 example above)!

To list a program on screen, type

```
LIST
```

and press **ENTER**. You may wish to list a program from a certain point onwards. This can be achieved by typing an appropriate line number after the **LIST** command. Try

```
LIST 15
```
(and **ENTER**)

to see this in action. If, at some point, you find you haven't left enough space between line numbers then you may use the edit menu to renumber a program. To do this, press the **EDIT** key then select the *Renumber* option from the menu that appears; this sets the gap between each line number to 10. Try this out and see how the line numbers change.

## REM, NEW, INPUT and GO TO

The command **NEW** erases any old programs and variables in the computer and starts the machine anew. Try it now; type:

```
NEW
```

and press **ENTER**. You'll see the *Welcome Screen* and then the *Startup menu*. With the menu on screen, select again the *NextBASIC* option.

Carefully type in this program, which changes Fahrenheit temperatures to Celsius:

```
10 REM Temperature Conversion
20 PRINT "deg F","deg C"
30 PRINT
40 @inpF:INPUT "Enter deg F",
   F
50 PRINT F, (F-32)*5/9
60 GO TO @inpF
```

Now run it. You will see the headings printed on the screen by line 20, but what happened to line 10? Apparently the computer has completely ignored it changing its colour to red. Indeed, REM in line 10 stands for REMark and is there solely to remind you of what the program does. A REM command consists of REM or the semicolon symbol ( ; ) followed by anything you like, and the computer will ignore it right up to the end of the line.

REM is not really part of a *NextBASIC* program; it just adds remarks to it for improved readability and documentation and gets totally ignored by *NextBASIC*. For example:

```
10 REM this is a remark
20 ; This is also a remark
```

are functionally equivalent, as are:

```
10 PRINT 10:REM Remark
20 PRINT 20:; Remark
```

Note that the colon (:) cannot be ommited, like:

```
10 PRINT 10;Remark
```

as then Remark forms part of the PRINT statement. A colon must ALWAYS be used to separate statements on the same line.

## Using STOP, BREAK and CONTINUE

By now, the computer has got to the INPUT command on line 40 and is waiting for you to type in a value for the variable F – you can tell this because at the bottom of the screen is a flashing cursor. Enter a number; remember to press ENTER afterwards! Now the computer has displayed the result and is waiting for another number. This is because of line 60, GO TO @inpF, which means exactly what it says. Instead of running out of program and stopping, the computer jumps back to line 40 where the label @inpF is located and starts again. So, enter another temperature. After a few more of these you might be wondering if the machine will ever get bored with this, it won't. Next time it asks for another number, enter the word stop. The computer will stay in the line and the cursor will change shape indicating a non acceptable entry. You have there the choice of hitting BREAK in which case you receive a report H STOP in INPUT, 40:2, which tells you why it stopped, and where (in the second statement of line 40, first being the label @inpF).

If you want to continue the program type:

```
CONTINUE
```

and the computer will continue with the INPUT line.

There's a synonym of CONTINUE which is really there for convenience and it's CONT. Try it in lieu of CONTINUE above; it will work in the same way.

Replace line 60 by GO TO 21 – it will make no perceptible difference to the running of the program. If the line number in a GO TO command refers to a non-existing line, then the jump is to the next line after the given number.

This, however, is NOT the case when using GO TO to jump to a *label* as the latter MUST exist otherwise an error will be produced. The same allowance for line numbers is true as well for RUN; in fact RUN on its own actually means RUN 0.

Now type in numbers until the screen starts getting full. When it is full, the computer will move the whole of the top half of the screen up one line to make room, losing the heading off the top. This is called scrolling.

When you are tired of this, stop the program as shown above and get the listing by pressing ENTER.

Look at the **PRINT** statement on line 50. The punctuation or *print modifier* in this – the comma ( , ) is very important, and you should remember that it follows much more definite rules than the punctuation in English. PRINT accepts 3 print modifiers: Commas ( , ), Semicolons ( ; ) and Apostrophes ( ' ).

Commas are used to make the printing start either at the left hand margin, or in the middle of the screen, depending on which comes next. Thus in line 50, the comma causes the Celsius temperature to be printed in the middle of the line. With a semicolon ( ; ) on the other hand, the next number or string is printed immediately after the preceding one. You can see this in line 50, if the comma is replaced by a semicolon. Note here that this is the exact reason why we need to enter a colon before the semicolon if we need to use it as a **REM**ark as discussed in the previous section!

Another punctuation mark you can use like this in **PRINT** commands is the apostrophe ( ' ). This makes whatever is printed next appear at the beginning of the next line on the screen but this happens anyway at the end of each **PRINT** command, so you will not need the apostrophe very much. This is why the **PRINT** command in line 50 always starts its printing on a new line, and it is also why the **PRINT** command in line 30 produces a blank line.

If you want to inhibit this, so that after one **PRINT** command the next one carries on on the same line, you can put a comma or semicolon at the end of the first. To see how this works, replace line 50 in turn by each of:

```
50 PRINT F,
50 PRINT F;
```

and:

```
50 PRINT F
```

and run each version – for good measure you could also try:

```
50 PRINT F'
```

The one with the comma spreads everything out in two columns, that with the semicolon crams everything together, that without either allows a line for each number and so does that with the apostrophe – the apostrophe gives a new line of its own, but inhibits the automatic one.

Remember the difference between commas and semicolons in **PRINT** commands; also, do not confuse them with the colons (:) that are used to separate commands in a single line. Now type in these extra lines:

```
100 REM this polite program
    remembers your name
110 INPUT n$
120 PRINT "Hello ";n$;"!"
130 GO TO 110
```

This is a separate program from the last one, but you can keep them both in the computer at the same time. To run the new one, type:

```
RUN 100
```

Because this program inputs a string instead of a number, it prints out two string quotes – this is a reminder to you, and it usually saves you some typing as well. Try it once with any alias you care to make up for yourself.

Next time round, you will get two string quotes again, but you don't have to use them if you don't want to. Try this, for example. Rub them out (with ⇨ and **DELETE** twice), and type:

```
n$
```

Since there are no string quotes, the computer knows that it has to do some calculation: the calculation in this case is to find the value of the string variable called n$, which is whatever name you happen to have typed in last time round. Of course, the INPUT statement acts like LET n$=n$, so the value of n$ is unchanged.

The next time round, for comparison, type:

```
n $
```

again, this time without rubbing out the string quotes. Now, just to confuse you, the variable n$ has the value "n$".

Generally speaking the INPUT parser[3] is very intelligent; with the exception of the example above, there is no way to insert an invalid or improperly formed entry during INPUT. For example moving the cursor back to the beginning of the line, using ⇦ and deleting the first set of quotes and pressing ENTER will produce the now familiar bleep sound and the program cursor will continue blinking until you correct the error by adding the first set of quotes again.

Now look back at that RUN 100 we had earlier on. That just jumps to line 100, so couldn't we have said GO TO 100 instead? In this case, it so happens that the answer is yes; but there is a difference. RUN 100 first of all clears all the variables and the screen, and after that works just like GO TO 100.

GO TO 100 doesn't clear anything. There may well be occasions where you want to run a program without clearing any variables; here GO TO would be necessary and RUN could be disastrous, so it is better not to get into the habit of automatically typing RUN to run a program.

Another difference is that you can type RUN without a line number, and it starts off at the first line in the program. GO TO must *always* have a line number or label.

Sometimes – by mistake – you write a program that you can't stop and won't stop itself. Type:

```
200 GO TO 200
RUN 200
```

This looks all set to go on for ever unless you pull the plug out; but there is a less drastic remedy. Press the BREAK key. The program will stop, saying L BREAK into program.
At the end of every statement, the program looks to see if these keys are pressed; and if they are, then it stops. The BREAK key can also be used when you are in the middle of using the cassette recorder or the printer, or various other bits of machinery that you can attach to the computer – just in case the computer is waiting for them to do something but they're not doing it. In these cases there is a different report, D BREAK - CONT repeats. CONTINUE, in this case (and in fact in most other cases too), repeats the statement where the program was stopped; but after the reports L BREAK into program or 9 STOP Statement, CONTINUE carries straight on with the next statement after allowing for any jumps to be made.

Run the name program again and when it asks you for input type:

n $ (after removing the quotes)

n$ is as of this moment an undefined variable and the computer will bleep as it doesn't recognise your inputted value as proper one. Use BREAK to get out of the program and then type:

3 *Parser is a computer program that reads data – usually in the foirm of a string of characters – analyses it and makes sure it conforms into a rigid syntax and/or set of rules in order to become meaningful to the computer*

```
n$="something definite"
```

(which has its own report of 0 OK, 0:1) and:

```
CONTINUE
```

you will find that you can use n$ as input data without any trouble.

In this case CONTINUE does a jump to the INPUT command in line 110. It disregards the report from the LET (implied in this case) statement because that said OK, and jumps to the command referred to in the previous report, the first command in line 110. This is intended to be useful. If a program stops over some error then you can do all sorts of things to fix it, and CONTINUE will still work afterwards.

As we said before, the report L BREAK into program is special because after it, CONTINUE does not repeat the command where the program stopped.

We've seen so far programs where execution jumps to the beginning with no graceful way of ending the program. What we're producing are called *never-ending loops* and are some of the great pitfalls a programmer can fall in. There are some cases where execution cannot be stopped (if for example we have disabled error reporting) or the BREAK key is inhibited. In these cases we have to provide with either a clear exit path to the program, or use a special keyword that ends a program prematurely and that keyword is STOP. Let's modify our polite program to be as follows:

```
100 REM this polite program
    remembers your name
110 INPUT n$
120 PRINT "Hello ";n$;"!"
130 STOP
```

and then give RUN. After we enter our name and the computer greets us, we'll get a 9 STOP statement, 130:1 report indicating we exited the program forcibly by the STOP command on line 130. We could have left line 130 out entirely and the program would have terminated with a 0 OK, 120:1 which would have indicated a proper program termination. In general, it's a good idea to provide exit paths in situations where the program may end up in a never-ending loop; *NextBASIC* provides us with such facilities as we're going to see further on.

## Error trapping

As we saw above, *NextBASIC* can occasionally generate error reports whether we have inadvertently caused them ourselves or because something went wrong. Sometimes we need our program to stop execution and other times we want it to recover from the error and continue (as it is the case above where we gave the CONTINUE command). For these cases, *NextBASIC* provides us with the ON ERROR command.

This can intercept (trap) any error report (except 0 OK which is not considered an error) thus allowing your programs to recover from *expected* error conditions.

Turning on error trapping is as simple as:

### ON ERROR *statementlist*

This will cause the statements contained in *statementlist* after the ON ERROR command to be executed whenever an error report would normally have been displayed. Note that this command must be part of a program and cannot be entered as a direct command.

To turn off error-trapping again, just use ON ERROR on its own.

This is required if you wish to generate errors again (and you may wish to do so if you need to know what went wrong). The following example will display **There was an error!** and terminate with the **9 STOP** statement error when line **20** is executed:

```
10 ON ERROR PRINT "There was
   an error!":ON ERROR:STOP
20 PRINT 5/0
```

ERROR *[n]*

To generate the last error that actually occurred (this does not need error-trapping to be turned off), just type the command:

```
ERROR
```

followed by **ENTER**. Assuming the program above, the following amendment will print the message but still give the correct **Number too big** report:

```
10 ON ERROR PRINT "There was
   an error!":ERROR
20 PRINT 5/0
```

Used with the optional *parameter n*, where *n* is a value between **0** and **3**, **ERROR** can return the error code (for *n*=**0**), the line (for *n*=**1**), the statement (for *n*=**2**) and the memory bank where it occured (for n=**3**) – See *Chapter 23* for details about memory banks. For example:

```
PRINT ERROR (1)
```

given after the example above would return **20**. Moreover there's also:

ERROR$

which prints the error report rather than just the code. You could modify the example above to be:

```
10 ON ERROR PRINT "There was
   error! "; ERROR$:ON
   ERROR:STOP
20 PRINT 5/0
```

which will print the actual error report **Number too big**. You could then substitute **STOP** with a **GO TO** to the line of error handling code without having to halt execution of your program.

An additional way with which you can obtain details of the last error and store them away maybe for purposes of statistical analysis is using the following command:

ERROR TO *codevar [, linevar, [statementvar, [bankvar]]]*

This will store the error code in the numeric variable *codevar*, the line number in *linevar*, the statement number in *statementvar* and the bank number in *bankvar (*do not worry about what *bank* means for the moment*)*. Note that you do not need to supply later variable names if you do not need the information, so all of these are valid:

```
ERROR TO e
ERROR TO e,l
ERROR TO e,l,s
ERROR TO e,l,s,b
```

For example, to get and store the error number into variable **e** and then print it but still stop execution, we could modify the first program as follows:

```
10 ON ERROR PRINT "There was
   an error!": ERROR TO e:
   PRINT e: ON ERROR:STOP
20 PRINT 5/0
```

If we allow the program to finish and then use **ERROR** we would have gotten the **9 STOP statement, 10:5** error report which would be the last error report in statement **5** of line **10** as **STOP** is considered an error. But by using **ERROR TO**, we'll get **6** printed on screen which is the error code for the **Number too big** error

So far we have seen the keywords **PRINT**, **LET**, **INPUT**, **RUN**, **LIST**, **GO TO**, **CONTINUE**, **STOP**, **ON ERROR**, **ERROR**, **ERROR$**, **ERROR TO**, **NEW** and **REM**. Apart from **ON ERROR** you can also enter them as direct commands – this is true of almost all commands in *NextBASIC*. **RUN**, **LIST**, **CONTINUE** and **NEW** are not usually of much use in a program, but they can be used regardless.

## Exercises

1. Put a **LIST** statement in a program, so that when you run it, it lists itself.

2. Write a program to input prices and print out the tax due (at 20 per cent). Put in **PRINT** statements so that the computer announces what it is going to do, and asks for the input price with extravagant politeness. Modify the program so that you can also input the tax rate (to allow for zero ratings or future changes).

3. Write a program to print a running total of numbers you input. (Suggestion: have two variables called total – set to **0** to begin with – and item. Input item, add it to total, print them both, and go round again.)

4. What would **CONTINUE** and **NEW** do in a program? Can you think of any uses at all for this?

# Chapter 2 – Decisions

## Making decisions

All the programs we have seen so far have been predictable; they went straight through the instructions, then went back to the beginning again. This is not very useful. In practice the computer would be expected to make decisions and act accordingly. There are three ways *NextBASIC* helps you make decisions: The first is by using the IF keyword in a short, medium and long format, the second by using the ON *case selection* keyword and the third is by using the *select operator* ? (question mark).

## Using IF… to make decisions

The short and medium form of IF are as follows:

IF *condition* THEN *action* [ELSE *alternative action*]

Clear the previous program from memory by using NEW and type in and run the following example:

```
10 REM Guess the number
20 INPUT "Enter the number to
   guess", a: CLS
30 INPUT "Guess the number",
   b
40 IF b=a THEN PRINT "That is
   correct": STOP
50 IF b<a THEN PRINT "That is
   too small, try again"
60 IF b>a THEN PRINT "That is
   too big, try again"
70 GO TO 30
```

You can see that in its simplest form an IF statement is:

IF *condition* THEN *action*

where *action* stands for a sequence of commands, separated by colons in the usual way. The condition is something that is going to be worked out as either true or false; if it comes out as true then the statements in the rest of the line after THEN are executed, but otherwise they are skipped over, and the program executes the next instruction.

The simplest conditions compare two numbers or two strings: they can test whether two numbers are equal or whether one is bigger than the other; and they can test whether two strings are equal, or (roughly) one comes before the other in alphabetical order. They use the relations =, <, >, <=, >= and <>. Additionally we can use NOT, AND, OR, ! (bitwise NOT), | (bitwise OR), & (bitwise AND), ^ | (bitwise XOR) all of which produce a *true* (1) or *false* (0) result

= means *equals*. Although it is the same symbol as the = in a LET command, it is used in quite a different sense.

< means *is less than* so that:

```
1 < 2
-2 <-1
-3 < 1
```

are all *true*, but:

```
1 < 0
0 <-2
```

are *false*.

> means *is greater than*, and is just like < but the other way round. You can remember which is which, because the thin end points to the number that is supposed to be smaller.

<= means *is less than or equal to,* so that it is like < except that it is true even if the two numbers are equal: thus 2<=2 is true, but 2<2 is false.

>= means *is greater than or equal to* and is similarly like >.

<> means *is not equal to*, the opposite in meaning to =.

The remaining operators are explained in length in *Chapter 6 – Expressions*.

Mathematicians usually write <=, >= and <> as ≤, ≥ and ≠. They also write things like 2<3<4 to mean 2<3 and 3<4, but this is not possible in *NextBASIC*.

Line 40 compares a and b. If they are equal then the program is halted by the STOP command. The report at the bottom of the screen 9 STOP, statement, 30:3 shows that the third statement, or command, in line 30 caused the program to halt, i.e. STOP.

Line 50 determines whether b is less than a, and line 60 whether b is greater than a. If one of these conditions is true then the appropriate comment is printed, and the program works its way to line 70 which tells the computer to go back to line 30 and start all over again. The CLS command in line 20 clears the screen to stop the other person seeing what you put in.

Note: in some versions of BASIC the IF statement can have the form:

IF *condition* THEN *line number*

This means the same as:

IF *condition* THEN GO TO *line number/label*

## ELSE

By adding the optional ELSE clause, more complex decisions can be made. This instructs the computer to run another set of commands if the IF...THEN test turns out to be false. It is important to note that, unlike some other implementations of BASIC, ELSE must follow a colon within a statement line; for instance:

```
IF number<0 THEN PRINT "Negative number":
ELSE PRINT "Positive number"
```

In the example above, if the condition is true (that is, the number is less than zero) then Negative number will be printed. If not, then Positive number will be printed on screen. But what if you for example wanted a third option to tell if the number is zero? You could use the ability to "nest" IF...THEN statements and use the ELSE clause to do so. Lets' re-write the above:

```
IF number<0 THEN PRINT "Negative number":
ELSE IF number>0 THEN PRINT "Positive
number": ELSE PRINT "The number is zero
```

You should see in the above that it is possible to execute a further IF...THEN statement if the condition in the original one was false. *NextBASIC* will work through the IF...THEN statements until it finds a condition that is true, and will execute that. If no conditions are true, then it will attempt to execute the final ELSE. More than one command can be executed within each part of an IF...THEN...ELSE statement also, so:

```
IF number<0 THEN PRINT "Negative number":
GO TO 100: ELSE IF number>0 THEN PRINT
"Positive number": GO TO 200: ELSE PRINT
"The number is zero" : zero += 1: GO TO
300
```

will allow you to jump to different parts of the program dependent on the results of the IF...THEN...ELSE statements; in this case, whether the number is negative, positive or zero (note that if the number is zero, one is added to the variable zero as well).

IF...THEN...ELSE however only works within a single program line and as a consequence it can be bulky and even somewhat unreadable. For that reason a longer form variation exists:

IF *condition* [ELSE [IF *condition*]] *action* END IF

It is immediately apparent that there is no THEN clause. This is what determines if it's the long, medium or short form of the IF structure. Medium and short forms use THEN and are thus single program line structures whereas the absence of THEN makes it the long form one. The long form IF has some specific syntax requirements. First off IF *condition* needs to be the first statement on its line and so are ELSE and ELSE IF. Secondly, the whole IF structure's command' sequence must be terminated by an END IF (which also needs to be the first statement on its line). Consider this example:

```
100  IF x>7:PRINT "x>7"
112     IF x>1000
113        PRINT "In fact it's
      huge"
114     ELSE
115        PRINT "But not too
      big"
116     ENDIF
120  ELSE IF x>3:PRINT "x>3 but
      x<=7"
140  ELSE IF x=3
150    PRINT "x=3"
160  ELSE
170    PRINT "x is too small to
      bother"
180  ENDIF
```

It vividly demonstrates how a combination of nesting IF...ELSE...ENDIF structures within an external superset of IF...ELSE...ENDIF can allow multiple decisions in an easy straightforward way. The main decision is contained in the initial IF on line 100 and the ensuing ELSE clause on line 160. There we're checking if x is greater than 7 or not. Obviously x can be smaller than 7 but how small is immaterial? This gets asked by the ELSE IF clauses on lines 120 and 140 where it's decided that for values from 3 (line 140) *up to and including* 7 (line 120) – the value *of greater than* 7 has been dealt with on line 100 as we saw – it does indeed matter. There is however a set of values that x can have that is *greater than* 7 and that's what the nested IF...ELSE...ENDIF structure on lines 112 through 116 allows us to test against where we further break it down to values *greater than* 1000 or *less or equal to* 1000 (but still greater than 7). The PRINT statements are there to demonstrate there's actions we can take per decision and could easily have been GO TO or INPUT or other sets of keywords actually doing something according to the value of x. Moreover, there are two styles of writing on display here. The compact one (lines 100 and 120) and the more relaxed with lots whitespace one (lines 140 to 180).

## Case selection with ON

Many BASIC dialects contain a special structure called SELECT CASE which is a specialised version of IF...ELSE IF...ENDIF that's much easier to use and read. *NextBASIC* has a somewhat similar keyword called ON. Its syntax is as follows:

ON *n*:<stmt0>:<stmt1>:<stmt2>:...:<stmtn>[: ELSE <statements>]

ON uses the resulting integer of rounding the value of expression *n* to the closest integer and executes the $n^{th}$ statement of the same line that follows it.

When the statement is executed, ON skips the rest of the line and moves to the next line, unless the statement was a non-returning jump such as GO TO, EXIT, RETURN or ENDPROC. If ON runs out of statements (in other words the integer part of n is greater than the number of statements that follows it, then, if the optional ELSE clause is present, all the statements that follow ELSE are executed otherwise the entire line is skipped. For example:

```
100 ON x:GO TO @xwaszero:GO TO
    @xwasone:STOP: GOSUB 200:
    ELSE BEEP  1,0: PRINT "x
    was > 3":STOP
110 PRINT "execution
    returned":STOP
200 PRINT "I'm in the
    subroutine now"
210 RETURN
```

It's easy to understand that if x is 2 then the program will terminate, if it's 0 it will jump to label @xwaszero, if it's 1 it will jump to label @xwasone, if it was 3 it will jump to the subroutine on line 200, return and continue at line 110 and in every other case the computer will beep and inform us that x was greater than 3 before finally halting. While at first sight this may not display much improvement over the nested IF...ELSE...END IF structure especially since it can only check for values greater than 0, in reality however you can prepare the variable to be checked ahead of time and simply test for that and moreover it is much simpler even typing-wise and more concise.

## Decisions with the ? operator

A final way of making decisions but this time within the confines of a single statement that accepts a parameter is provider via the *select operator* ? (question mark). This has the following form:

*n*?(*expr0,expr1,expr2...*)

If this reminds you a bit of the ON keyword syntax above you'd be correct as based on the value of n the $n^{th}$ expression on the list is evaluated. If you run out of expressions in the list then the last expression is always evaluated. The select operator can take either numeric (be it floating point or integer) or string values in its evaluation list, however they all have to be the same. Although it's not immediately apparent how one could use it in te decision-making process, type and RUN the following example which will make it all clear:

```
10 INPUT "Enter a number "; n
20 PRINT n?(%5, %10, %20)
30 PRINT n?("n is zero", "n is
   one", "n is two", "n is
```

```
        three", "n is four or
        more")
    40 GO TO n?(100, 60, 120, 130,
        10)
    50 STOP
    60 PRINT "n was one": STOP
   100 PRINT "n was zero": STOP
   120 PRINT "n was two": STOP
   130 PRINT "n was three": STOP
```

(Do not mind the % symbols in the list of line 20; we will learn more about them in *Chapter 6 – Expressions*). As you enter values in the program, you will see how your input affects both the PRINT statements but also (and here's the important part) the GO TO statement on line 40 in effect creating a *conditional* GO TO. Moreover, instead of having to type 5 different GO TO statements like we would have in the case of IF...ELSE...END IF and ON, we only have to type one making it a great time-saver.

Obviously the ability of the *select operator* to take either *string* or *numeric expressions* as parameters makes it useful for many things like for example a *conditional* LET apart from changing the flow of our program, so other than just line numbers we just used, we can make it calculate and further enhance our decision making as we'll see in *Chapter 6*.

# Chapter 3 – Looping

## Using FOR, TO and NEXT

Suppose you want to input five numbers and add them together. One way (don't type this in unless you are feeling dutiful) is to write:

```
 10  total=0
 20  INPUT a
 30  total+=a
 40  INPUT a
 50  total+=a
 60  INPUT a
 70  total+=a
 80  INPUT a
 90  total+=a
100  INPUT a
110  total+=a
120  PRINT total
```

This method is not good programming practice. It may be just about controllable for five numbers, but you can imagine how tedious a program like this to add ten numbers would be, and to add a hundred would be just impossible.

Much better is to set up a variable to count up to 5 and then stop the program, like this (which you should type in):

```
10  total,count=0,1
20  INPUT a
30  REM count=number of times
    that a has been input so
    far
40  total += a
50  count += 1
60  IF count <= 5 THEN GO TO 20
70  PRINT total
```

Notice how easy it would be to change line 60 so that this program adds ten numbers, or even a hundred.

This sort of counting is so useful that there are two special keywords to make it easier: FOR and NEXT that are always used together. Using these, the program you have just typed in does exactly the same as:

```
10  total = 0
20  FOR count = 1 TO 5
30    INPUT a
40    ;count=number of times
    that a has been input so
    far
50    total += a
60  NEXT count
```

```
80 PRINT total
```

The variable named count, is called the *control variable* of a FOR … NEXT loop.

The effect of this program is that count runs through the values 1 (the *initial value*), 2, 3, 4 and 5 (the *limit*), and for each one, lines 30, 40 and 50 are executed. Then, when count has finished its five values, line 80 is executed.

STEP

The control variable, does not have to increase by 1 each time; you can change this 1 to anything you like by adding a STEP clause in the FOR command. The most general form for a FOR command is:

FOR *control variable = initial value* TO *limit* STEP *step*

where the *initial value*, *limit* and *step* are all *numeric expressions*; things in other words that the computer can calculate as numbers – like the actual numbers themselves, or sums, or the names of numeric variables. So, if you replace line 20 in the program by:

```
20 FOR count=1 TO 5 STEP 3/2
```

then count will run through the values 1, 2.5 and 4. Notice that you don't have to restrict yourself to whole numbers, and also that the control value does not have to hit the limit exactly – it carries on looping as long as it is less than or equal to the limit. Try this program, to print out the numbers from 1 to 10 in reverse order.

```
10 FOR n=10 TO 1 STEP -1
20   PRINT n
30 NEXT n
```

We have said before that the program carries on looping as long as the control variable is less than or equal to the limit. If you work out what this would mean in this case, you will see that it gives nonsense. The normal rule has to be modified; when the step is negative, the program carries on looping as long as the control variable is greater than or equal to the limit.

You must be careful if you are running two FOR...NEXT loops together, one inside the other. Try this program, which prints out the numbers for a complete set of six spot dominoes.

```
10 FOR m=0 TO 6
20   FOR n=0 TO m
30     PRINT m;":";n;" ";
40   NEXT n
50   PRINT
60 NEXT m
```

n-loop

m-loop

You can see that the n-loop is entirely inside the m-loop – they are properly *nested*. What must be avoided is having two FOR ... NEXT loops that overlap without either being entirely inside the other, like this:

```
 5 REM this program is wrong
10 FOR m=0 TO 6
20   FOR n=0 TO m
30     PRINT m;":";n;" ";
40   NEXT m
50   PRINT
60 NEXT n
```

m-loop

n-loop

Two FOR ... NEXT loops must either be one inside the other, or be completely separate.

Another thing to avoid is jumping into the middle of a FOR … NEXT loop from the outside. The control variable is only set up properly when its FOR statement is executed, and if you miss this out the NEXT statement will confuse the computer. You will probably get an error report saying NEXT without FOR or Variable not found.

There is nothing whatever to stop you using FOR and NEXT in a direct command. For example, try:

```
FOR m=0 TO 10: PRINT m: NEXT m
```

You can sometimes use this as a (somewhat artificial) way of getting round the restriction that you cannot GO TO anywhere inside a command – because a command has no line number. For instance:

```
FOR m=0 TO 1 STEP 0: INPUT a: PRINT a:
NEXT m
```

The step of zero here makes the command repeat itself forever.

This sort of thing is not really recommended, because if an error crops up then you have lost the command and will have to type it in again –and CONTINUE will not work.

For additional speed and efficiency, *NextBASIC* also allows integer variables to be used as the index in FOR … NEXT, eg:

```
10 FOR %i=%$c9 TO 220
20   PRINT %i
30 NEXT %i
```

Integer loops run much faster than loops using a standard floating point control variable so they're preferred, especially where speed is a concern. Remember, however, that there is only a limited amount of integer variables, so a small bit of planning is warranted before starting with your program.

## EXIT

Sometimes we need to prematurely exit from a loop, be it a FOR...NEXT (See previous section) or REPEAT...REPEAT UNTIL (See next section) one. There is a seemingly obvious solution to that; jump out of the loop with GO TO however this is ill advised as GO TO doesn't exit a loop "cleanly" and therefore should not be used. Instead, there's a specialised command however that allows for a "clean" exit and that is (the aptly named):

EXIT [*n*]

where *n* is an optional line number or label to jump to. EXIT on its own will jump to the next statement after the end of the loop. Consider this example (the actual syntax of the loop is explained in the next section):

```
100 REPEAT
110   INPUT n
120   IF n=33 THEN EXIT 150
130 REPEAT UNTIL n<0
140 PRINT "Loop ended normally"
150 PRINT "Loop ended early"
```

The loop above will terminate normally (when the condition set on line 130 is satisfied) when you enter a negative value but early if you input enter the value 33. EXIT can also be used to get out of nested loops using successive EXIT statements on the same line within

the innermost loop. Note that in such cases, only the final EXIT statement can take the optional parameter. To illustrate, consider the following example:

```
100 FOR i=1 TO 10
110   FOR j=1 TO 10
120     PRINT i,j
130     IF j*i>80 THEN EXIT:EXIT 170
140   NEXT j
150 NEXT i
160 STOP
170 PRINT "Product exceeded 80":STOP
```

Note that when in a loop that exists within a procedure and/or subroutine, it is acceptable to use ENDPROC or RETURN as a legitimate way to exit said loop.

## REPEAT ... REPEAT UNTIL loops

*NextBASIC* has another way of looping: a set of commands (or rather a single command block) called REPEAT … REPEAT UNTIL. You will have noticed that FOR … NEXT relies on counting to control the loop however you can also use a condition to control a loop. This type of loop begins with a REPEAT statement to indicate the beginning of the loop and a REPEAT UNTIL statement at the end, which also contains the condition to exit the loop. Try this:

```
10 REPEAT
20   INPUT "Enter a number,
   or enter -1 to stop > ";n
30   PRINT n
40 REPEAT UNTIL n=-1
50 PRINT "Thank you!"
```

This program will keep accepting numbers and printing them, until you type -1 when it will politely thank you for your numbers. In a REPEAT … REPEAT UNTIL loop, everything between the REPEAT and the REPEAT UNTIL command will be executed (in this case, this would be lines 20 and 30), until the condition in the REPEAT UNTIL statement becomes *true* (in this case, that the number you have entered is -1). Note that because the condition is checked at the end, the block of statements will always execute at least once.
The following, for example, would print an erroneous statement:

```
10 x=1
20 REPEAT
30   PRINT "x is ";x;" but isn't 1"
40 REPEAT UNTIL x=1
50 PRINT "x is now 1."
```

Because line 30 is executed before the condition is checked at line 40, the message x is 1, but it isn't 1 will still be printed, which is clearly wrong. Like a FOR … NEXT loop, you can also nest REPEAT loops, if you need to. So:

```
10 n=1
20 REPEAT
30   PRINT "Counting to ";n
40   c=1
50   REPEAT
60     PRINT c;", ";
```

```
 70       c+=1
 80    REPEAT UNTIL c>n
 90    PRINT "I'll count a bit
    higher"
100    n+=1
110 REPEAT UNTIL n=10
120 PRINT "OK, I'm done now"
```

will work fine – try it and see if you can see what is happening. You can also make a REPEAT loop continue indefinitely, if you use a zero in the REPEAT UNTIL statement. Type in this program:

```
10 REPEAT
20    PRINT "Hello world!"
30 REPEAT UNTIL 0
```

It will continue printing Hello world! to the screen, stopping only to ask if you want to scroll (unless you press the BREAK key, of course). Why? Zero can be seen in *NextBASIC* as *false* when used in this way, so the REPEAT UNTIL 0 statement will always give a *false* result; hence the loop will continue indefinitely. Obviously you can exit such a loop with EXIT if need be

## WHILE

The WHILE command, used within a REPEAT loop, can provide an alternative way of leaving the loop before reaching the REPEAT UNTIL statement. If the condition in the WHILE statement is *true*, the loop continues. But if it is *false*, then the remaining statements in the loop will be ignored, the loop will be exited and the program will resume with the line after the REPEAT UNTIL statement. Try this:

```
10 REPEAT
20    INPUT "Enter a number, or
    enter a negative number to
    stop > ";n
30    WHILE n>=0
40    PRINT n
50 REPEAT UNTIL 0
60 PRINT "Thank you!"
```

It is a different approach to the example seen earlier, this time using WHILE to check the number entered (and also accepting any negative number to stop). WHILE can also be used to exit a loop before any statements are executed, should you need to. Try:

```
10 y=0
20 REPEAT : WHILE y<22
30    PRINT AT y,0;"This is
      line ";y;"."
40    y+=1
50 REPEAT UNTIL 0
```

You will note that when y reaches 22, the loop will exit before printing the line number. It should also be pointed out that not only can you place a WHILE anywhere within the loop, but you can also place more than one WHILE in the same loop, if you have different conditions to check to leave the loop.

## Error trapping within REPEAT … REPEAT UNTIL loops

Error trapping within **REPEAT … REPEAT UNTIL** loops as well as within *subroutines* and *procedures* is localised. Refer to the last section of *Chapter 4 – Localised Error Trapping* for a complete example that covers all cases of error trapping in these programming structures.

## Exercises

1. A control variable has not just a name and a value, like an ordinary variable, but also a limit, a step, and a reference to the statement after the corresponding **FOR** statement. Persuade yourself that when the **FOR** statement is executed all this information is available (using the initial value as the first value the variable takes), and also that this information is enough for the **NEXT** statement to know by how much to increase the value, whether to jump back, and if so where to jump back to. Run the third program above and then type:

   ```
   PRINT count
   ```

   Why is the answer **6**, and not **5**? (Answer: the **NEXT** command in line 60 is executed five times, and each time **1** is added to **count**. The last time, **count** becomes **6**; and then the **NEXT** command decides not to loop back, but to carry on, **count** being past its limit.)

2. What happens if you put **STEP 2** in line 20?

3. Change the third program so that instead of automatically adding five numbers, it asks you to input how many numbers you want adding. When you run this program, what happens if you input 0, meaning that you want no numbers adding? Why might you expect this to cause problems for the computer, even though it is clear what you mean? (The computer has to make a search for the command **NEXT count**, which is not usually necessary.) In fact this has all been taken care of.

4. In line 10 of the fourth program above, change **10** to **100** and run the program. It will print the numbers from **100** to **79** on the screen, and then say **scroll?** at the bottom. This is to give you a chance to see the numbers that are about to be scrolled off the top. If you press **n**, **BREAK** or the **space bar**, the program will stop with the report **D BREAK - CONT repeats**. If you press any other key, then it will print another 22 lines and ask you again.

5. Delete line 30 from the fourth program. When you run the new curtailed program, it will print the first number and stop with the message **0 OK**. If you type:

   ```
   NEXT n
   ```

   The program will go once round the loop, printing out the next number.

6. Refer back to the example in the **REPEAT UNTIL** section, where the message **x is 1, but it isn't 1** was displayed incorrectly. Rewrite this using **WHILE** so that the message does not appear when x is indeed 1. Change the value of **x** in line 10 to check this works correctly.

# Chapter 4 – Procedures and Subroutines

## Branching

As we already saw, in the course of a program we may need to jump somewhere else inside the program. So far we've seen the GO TO keyword that does just that. GO TO however has disadvantages: If the code jumped to by GO TO needs to be used by some other portion of the program and then return to where it was called, you would need to either introduce complex code to find where the jump came from as to use an appropriate GO TO command to return to it or copy the code multiple times to be called individually by each location.

## GO SUB and RETURN

To account for this deficiency, *NextBASIC* has the statements GO SUB (GO to SUBroutine) and RETURN which are used together. Reusable code called upon multiple times within a program is known as a *subroutine*, and it can be used – or *called* – from anywhere else in the program without having to type it again or remember where the call came from. The call to a subroutine takes the form:

GO SUB *n*

where *n* is the line number or label of the first line in the subroutine. It is just like GO TO *n* except that the computer remembers where the GO SUB statement was so that it can come back again after doing the subroutine. It does this by putting the line number and the statement number within the line (together these constitute the *return address*) on top of a pile of them (the *NextBASIC return stack* – see *Chapter 23* for details):

The command

RETURN

takes the top return address off the GO SUB stack, and goes to the statement after it. As an example, let's look at the number guessing program again. Retype it as follows:

```
 10 REM A rearranged guessing
    game
 20 INPUT a: CLS
 30 INPUT "Guess the number ",b
 40 IF a=b THEN PRINT
    "Correct": STOP
 50 IF a<b THEN GO SUB 100
 60 IF a>b THEN GO SUB 100
 70 GO TO 30
100  PRINT "Try again"
110 RETURN
```

The GO TO statement in line 70 is very important because otherwise the program will run on into the subroutine and cause an error (7 RETURN without GO SUB) when the RETURN statement is reached.

Here is another rather silly program illustrating the use of GO SUB:

```
100 x=10
110 GO SUB 500
120 PRINT s
130 x+=4
```

```
140 GO SUB 500
150 PRINT s
160 x+=2
170 GO SUB 500
180 PRINT s
190 STOP
500  s=0:; This is the start
     of the subroutine
510  FOR y=1 TO x
520    s+=y
530  NEXT y
540 RETURN
```

When this program is run, see if you can work out what is happening. The subroutine starts at line 500.

A subroutine can happily call another, or even itself (a subroutine that calls itself is *recursive*), so don't be afraid of having several layers.

## LOCAL keyword

LOCAL is a special keyword reserved only for subroutines (see above) and procedures (see below) which ensures that the variables that follow it, are independent of the rest of the program and only valid for the duration of the execution of the subroutine or procedure. Its syntax is as follows:

LOCAL *var*[=*value*]

wher *var* is any variable type (string, numeric or integer) or array and value is the initial (or *default*) value for said variable. Local arrays[1] need to be dimensioned separately after being localised by LOCAL. Multiple variables can be declared in the same LOCAL statement, separated by commas and there can be any number of LOCAL statements in a subroutine or procedure as long as there is enough memory for them.Consider the following example (which won't make much sense until you reach the *procedures section* further below):

```
100 PROC x(a(),size,2) TO
    result: PRINT result:STOP
110 DEFPROC x(input(),size,y)
120   LOCAL z():LOCAL tot=0
130   DIM z(size)
150   FOR i=1 TO size:z(i)=input(i)*
    y: NEXT i
160   FOR i=1 TO size: tot+=z(i):NEXT i
170 ENDPROC = tot
```

The moment that branching back occurs, local variables are released. Consider this silly example (which also demonstrates again the initialisation of local a$):

```
 10 a$ = "Test"
 20 GO SUB 100
 30 PRINT a$
```

1 *Except integer arrays which are predimentioned*

```
 40 STOP
100   LOCAL a$="Different Value"
110   PRINT a$
120 RETURN
```

This will print Different Value and Test on your screen as LOCAL creates in a sense *two* versions of a$. The second one exists only until the RETURN keyword is reached.

## PRIVATE and PRIVATE CLEAR

PRIVATE is similar to LOCAL in the fact that's reserved only for *subroutines* (see above) and *procedures* (see next section). Its syntax is as follows:

PRIVATE *var*[=*value*]

where *var* is a numeric variable (Unlike LOCAL, only numeric variables are accepted by PRIVATE) and *value* is the optional initialisation value. The latter is important as you will see below. PRIVATE ensures that the variables that follow it, are both independent of the rest of the program and that they are retained every time the *procedure* is called. In order to reset the value of all private variables (to 0 or to the value initialised to by the PRIVATE statement) we can use PRIVATE CLEAR. When calleded from within *banked code* (See *Chapter 23 – The Memory* for information about banks and banked code) PRIVATE CLEAR initialises only the private variables for the procedures in that bank otherwise the private variables for the main program are reset. A small example on how PRIVATE and PRIVATE CLEAR work is the following:

```
100 PRIVATE CLEAR
110 FOR i=1 TO 10: PROC
    iterate():NEXT i:STOP
120 DEFPROC iterate()
130   PRIVATE callcount=0
140   callcount+=1
150   PRINT "I have been called
    ";callcount;"times"
160 ENDPROC
```

## Procedures (DEFPROC / ENDPROC / PROC)

Procedures are a special form of subroutines. Imagine them as a combination of subroutines and functions (See *Chapter 8 – Functions*). Like subroutines and the GO TO keyword they branch execution to a different segment of the program to better organise and reuse code. Unlike subroutines but like functions, they can accept multiple variables as *parameters*, can be named and when called they do not require a line number or label.
Think of procedures as a way to extend *NextBASIC* commands much like user defined functions extend the inbuilt functions (See *Chapter 8* for more details).

Procedure *parameters* can be regular numeric, integer and string variables as well as arrays which follow all the naming conventions of the former (As seen in *Chapters 1 – Basic Programming Concepts* , *7 – Expressions* and *11 – Arrays*).

Procedures can carry meaningful names following the naming conventions of numeric variables (See *Chapter 7 – Expressions* for valid numeric variable names), while they can also optionally be followed by a $ sign. The latter has no effect in functionality but can be used to identify what the procedure does (for example manage strings)

Procedures are defined by the keywords DEFPROC which takes the form:

DEFPROC *name[$]* (*[parameter1[=def1][,...[, parameterN[=defN]]]*)

and **ENDPROC** which takes one of two forms:

**ENDPROC**

or –optionally–

**ENDPROC** =*result1[,...,resultN]*

Anything that follows the keyword **DEFPROC** is the procedure itself, however there can be multiple exit points for each procedure designated by separate **ENDPROC** statements.

Parameters in *brackets*, denote that the syntax is optional and *def1* to *defN* (also in *brackets*) are optional default values for each parameter. Procedures are called with the keyword **PROC** (and **BANK PROC** in the case of a banked procedure). This, like **ENDPROC** takes two forms:

[**BANK** *n*] **PROC** *name* ([*parameter1[,...[,parameterN]]]*)

which calls the procedure named **name** with optional parameters 1 through N –or–

[**BANK** *n*] **PROC** *name* (*[parameter1[,...[,parameterN]]]*) **TO** *variable1*[*,...,variableN] which is the same as above but assigns the values returned by the procedure to the optional variables 1 through N.*

Note that if there's a default value for a parameter, we can omit it when calling the procedure with **PROC**.

**Notes**

If you're using the *NextBASIC*'s memory bank management facilities to extend the size of your programs, the following apply:

1. Any **GO TO**, **PROC** or **GO SUB** within a banked section will go to a line or label in the same bank.
2. Any **RETURN** will always return to the calling bank.

Consider the example below:

```
10 CLS
20 PROC Pdemo(11): PROC
   HelloWorld("Hello
   World!",1)
30 PROC HelloWorld("Hello
   Stop!",):; 2nd parameter
   omitted
40 GO TO 160
50 DEFPROC Pdemo(x)
60   PRINT x; " raised to the
   2nd power is:"; x*x
70 ENDPROC
80 DEFPROC HelloWorld(z$,
   n=0)
90   LOCAL a$,l
```

```
100    IF n=0 THEN PRINT z$:
       ENDPROC
120    IF n=1 THEN LET l=LEN z$
130    a$=z$(l)+z$+z$(l)
140    PRINT z$' INVERSE 1; a$
150 ENDPROC
160 STOP
```

As you can see, there's a default value for parameter *n* and two separate exit points for procedure HelloWorld: one at line 100 and one at line 150. Line 40 is mandatory or rather a condition to jump over the procedures defined is mandatory as without it, after execution of both procedures the next available line would have been 50. DEFPROC can only appear in a program line. Attempting to define a procedure interactively will result in the error Direct Command Error. Also, supplying the wrong type of variable as a parameter (ie. a string instead of a number) will result in the error: Q Parameter error.

Executing the program will return the following:

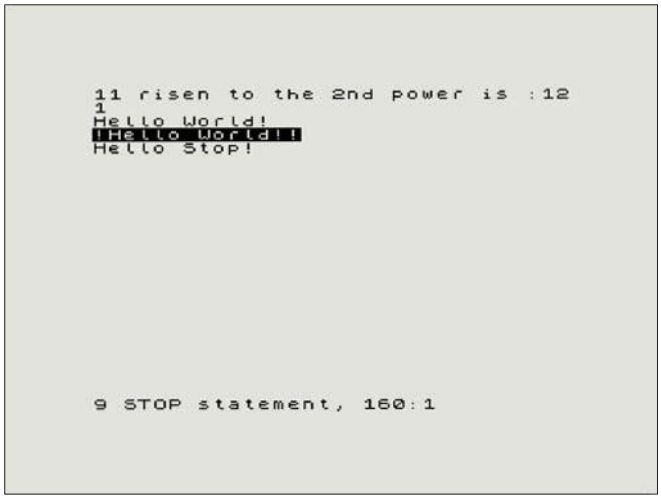



*Fig. 3 - Screen output from the example procedures*

As we saw in the definition of the DEFPROC, ENDPROC and PROC keywords as well as the examples above, there are optional parameters that can be passed to procedures when called with the results of the procedures' execution being assigned to multiple variables at the time. Consider this example that calculates the factorial of a number:

```
 10 INPUT "Enter a number
    1+:";x
 20 IF x>33 THEN PRINT "Your
    Next cannot handle this
    number!": GO TO 999
 30 PROC factorial(x) TO f
 40 IF f>0 THEN PRINT "The
    factorial of ";x;" is ";f:
    ELSE GO TO 999
999 STOP
1000 DEFPROC factorial(n)
```

```
1010 IF n<0 OR n<> INT n THEN
     PRINT "Factorial only
     possible for 0 or positive
     integers":ENDPROC =
     -1:ELSE IF (n = 0 OR n=1)
     THEN ENDPROC =1
1020 LOCAL partial
1030 PROC factorial(n-1) TO
     partial
1040 ENDPROC =n*partial
```

Apart from being a good example of *recursion*[2] we can see how this procedure feeds itself the results of the previous iteration via the local variable **partial**. Each iteration reduces the value by **1** as evidenced in line 1030. There's an obvious extra iteration that could be skipped when **n** becomes **1** but it's not important for the purpose of this example.

When calling a procedure with the **PROC ... TO...** version of the **PROC** keyword, **ENDPROC** must use the optional form **ENDPROC =result1...** and have as many results returned (separated by commas) as the calling **PROC** requested. **PROC** may be called without a **TO** or with a partial list of the result variables returned by **ENDPROC** but the inverse cannot happen and will return error **Q Parameter error**. For example this program:

```
10 product = 0
20 PROC mul(3) TO product
30 PRINT product
40 STOP
50 DEFPROC mul(x)
60   LOCAL a
80   a=x*2
90 ENDPROC =a
```

will return **6** when run. When we change line 20 to read:

```
20 PROC mul(3)
```

it will return **0** as variable **product** hasn't been changed from its initial assignment, however if we return line 20 to its original form and change line 70 to:

```
70 ENDPROC
```

then execution of the program will produce a **Q Parameter error**.

## Passing parameters by reference with REF

Normally calling a procedure with parameters is done *by value*: this means that the value of the parameter is passed on to the procedure and any changes that occur are invisible to the caller. Passing *by reference* on the other hand lets the caller be aware of the changes.

This is mainly intended for arrays, as the default (by value) will be slower and more memory intensive, but it can also be useful for strings. Numeric variables, are faster if passed by value so passing them by reference is not recommended. To make a parameter a reference we use the **REF** keyword within the **DEFPROC** parameter block; such parameters must be passed by the calling **PROC** as a variable or array name only. It is not permitted to have default values for **REF**erenced parameters. For example:

2 *The ability of the code to call itself*

```
100 PROC tl$(x$(),5):STOP
110 DEFPROC tl$(REF
    input$(),index=1)
120 input$(index) =
    input$(index)(2 TO)
130 ENDPROC
```

On the example above index is not passed by reference and therefore it can have a default value. Note that integer variables and arrays cannot be passed by reference!!!

## Reading parameters with DATA and READ

A PROC may call a DEFPROC with more parameters than the DEFPROC requires. In this case, if the DEFPROC has the keyword DATA as its final parameter, the procedure may read the additional parameters one at a time using READ. The new function DATA (see also *Chapter 5*) can be used to determine if there are further PROC parameters left to read. As an example:

```
100 PROC printem("Digits",0,1,2,
    3,4,5,6,7,8,9):STOP
110 DEFPROC printem(name$, DATA)
115    LOCAL n
120 PRINT name$
130   REPEAT:WHILE DATA
140 READ n:PRINT n
150 REPEAT UNTIL 0
160 RETURN
```

## Trapping errors locally

As well as (or instead of) having a global error-trapping routine for your program as exhibited at the end of *Chapter 1, each procedure, subroutine and repeat loop may have its own local error-trapping routine, simply by using the* ON ERROR command within it.

When an error occurs within a repeat loop, subroutine or procedure, it will be trapped by its own ON ERROR routine if there is one. If not, the error will be passed out to the next level and trapped by any ON ERROR routine there and so on. Only if there is no ON ERROR at any level above the command that caused the error will a normal error report be generated. For example:

```
 10 ON ERROR PRINT "Outer error
    handler!":ERROR
 20 REPEAT
 30   PRINT "Starting..."
 40   ON ERROR PRINT "Oops!":ON
    ERROR:STOP
 50   GO SUB 100
 60   PRINT "Iterating..."
 70   ON ERROR
 80 REPEAT UNTIL 0
 90 STOP
100 ON ERROR PRINT "Bad
    pigs!":RETURN
```

```
110 PROC myproc()
120 PRINT "Pigs:";pigs
130 RETURN
200 DEFPROC myproc()
210   LOCAL m
220   ON ERROR PRINT "Myproc
    died...":ENDPROC
230   PRINT "m=";m,"n=";n
240 ENDPROC
```

Note that any LOCAL commands in a procedure or subroutine must come before a local error handler (ie lines 210 and 220 in the example cannot be reversed).

# Chapter 5 – READ, DATA, RESTORE

## READ, DATA and RESTORE

In some previous programs we saw that information, or data, can be entered directly into the computer using the INPUT statement. Sometimes this can be very tedious, especially if a lot of the data is repeated every time the program is run. You can save a lot of time by using the READ, DATA and RESTORE commands. For example:

```
10 READ a,b,c
20 PRINT a,b,c
30 DATA 10,20,30
```

A READ statement consists of READ followed by a list of the names of variables, separated by commas. It works rather like an INPUT statement, except that instead of getting you to type in the values to give to the variables, the computer looks up the values in the DATA statement.

Each DATA statement is a list of expressions – numeric or string expressions separated by commas. You can put them anywhere you like in a program, because the computer ignores them except when it is doing a READ. You must imagine the expressions from all the DATA statements in the program as being put together to form one long list of expressions, the DATA list. The first time the computer goes to READ a value, it takes the first expression from the DATA list; the next time, it takes the second; and thus as it meets successive READ statements, it works its way through the DATA list. (If it tries to go past the end of the DATA list, then it gives an error. See the next section for an easy way to avoid that)

Note that it's a waste of time putting DATA statements in a direct command, because READ will not find them. DATA statements have to go in the program. Let's see how these fit together in the program you've just typed in. Line 10 tells the computer to read three pieces of data and give them the variables a, b and c. Line 20 then says PRINT these variables. The DATA statement in line 30 gives the values of a, b and c. To see the order in which things work change line 20 to:

```
20 PRINT b,c,a
```

The information in DATA can be part of a FOR...NEXT loop. Type in:

```
10 FOR n=1 TO 6
20 READ d
30 DATA 2,4,6,8,10,12
40 PRINT d
50 NEXT n
```

When this program is RUN you can see the READ statement moving through the DATA list. DATA statements can also contain string variables. For example:

```
10 READ dat$
20 PRINT "The date is",dat$
30 DATA "January 1st, 2024"
40 STOP
```

This is the simple way of fetching expressions from the DATA list; start at the beginning and work through until you reach the end. However, you can make the computer jump about in the DATA list, using the RESTORE statement. This has RESTORE, followed by a line number, and makes subsequent READ statements start getting their data from the

first DATA statement at or after the given line number. (You can miss out the line number, in which case it is as though you had typed the line number of the first line in the program.)

Try this program:

```
10 READ a,b
20 PRINT a,b
30 RESTORE 10
40 READ x,y,z
50 PRINT x,y,z
60 DATA 1,2,3
70 STOP
```

In this program the data required by line 10 made a=1 and b=2. The RESTORE 10 instruction allowed x, y and z to be READ starting from the first number in the DATA statement. RUN this program again, without line 30 and see what happens.

**Notes**

You can store DATA statements in memory banks to take advantage of the expanded memory available on the ZX Spectrum Next. Refer to *Chapter 23 – The Memory*, for information on how to use BANK RESTORE to do this.

READ, DATA and RESTORE accept integer variables following the conventions set forth in *Chapter 6 – Expressions*.

## DATA (function)

Apart from its use in subroutines as we saw in the previous chapter, DATA can also be used as a *function* that returns what the next *DATA* item to be read is; Return values can be: *numeric* (1), *string* (2) or *none* (0) – when there are no more data items available. This is most useful if we want to load a list of items with varying data types. Consider this example:

```
10 @rd:ON DATA:GO TO @prn:
   READ number: READ text$:
   ELSE GO TO @rd
20 GO TO @rd
30 @prn: PRINT text$;number
40 DATA 3,"Here we come KS"
```

What have we done here? We've used the handy ON...ELSE *conditional selection* structure and placed the DATA *function* in lieu of a variable argument since it returns one of 3 values. Then used the data type look ahead that DATA as a function provides to decide whether our next READ operation will be a numeric or string one (As ON expects its statements according to an increasing order of values, the *exit condition* is first).

Note that ELSE is really redundant here as there are 3 possible values of DATA and all are covered by the statements available, however it was put there as an additional demonstration of the complete ON...ELSE structure.

# Chapter 6 – Expressions

## Mathematical operations +, - , * , /, MOD

You have already seen some of the ways in which the ZX Spectrum Next can calculate with numbers. It can perform the four arithmetic operations +, -, * and / (remember that * is used for multiplication, and / is used for division), and it can find the value of a variable, given its name. The example:

```
tax=sum*20/100
```

gives just a hint of the very important fact that these calculations can be combined. Such a combination, like sum*20/100, is called an *expression*; so an *expression* is just a short-hand way of telling the computer to do several calculations, one after the other. In our example, the *expression* sum*20/100 means *look up the value of the variable called "sum", multiply it by 20, and divide the result by 100*.

There's also one more mathematical operation, the *modulo* which returns the *remainder* of a division. It is used in the same way as the division operator but is denoted instead by MOD. As an example the direct commands:

```
PRINT %17 MOD 6
PRINT 17 MOD 6
```

will both return 5 which is the remainder of the division of 17 by 6; note the percent symbol (%) that prefixes 17 in the first example: this is what defines it as an *Integer Expression* – We will look at this in a little bit.

## Order of mathematical calculations

It's important here to underline the order in which mathematical expressions are evaluated: For floating point operations, multiplications and divisions are done first. They have higher priority than addition and subtraction. Relative to each other, multiplication and division have the same priority, which means that the multiplications and divisions are done in order from left to right. When they are dealt with, the additions and subtractions come next; these again have the same priority as each other, so we do them in order from left to right. This is very important because one can arrive to a wrong result if they forget said order.

Although all you really need to know is whether one operation has a higher or lower priority than another, the computer does this by having a number between 1 and 16 to represent the priority of each operation: * and / have priority 8, and + and - have priority 6.

This order of calculation is absolutely rigid, but you can circumvent it by using parentheses; anything in parentheses is evaluated first and then treated as a single number.

**Notes**

Specifically for *Integer Expressions*, the order of calculations is strictly left-to-right with the exception of the use of parentheses. In the case of multiple sets of parentheses, their contents are also evaluated from left-to-right

## Bitwise, relational and logical operators

Within every kind of expression there's a number of bitwise, relational and logical operations that can be performed other than simple mathematical operations. Specifically for numbers these can be performed on floating point or integer numbers. While the operators are the same, there are subtle (and some not so subtle) differences in the way things work and we'll attempt to show these below.

In addition to the above there's a unary operator which follows:

## Unary/Bitwise NOT (!)

As we discussed in the previous section *NextBASIC* provides one additional unary operator, which is an operator that requires one number alone. This is:

| | |
|---|---|
| ! | bitwise NOT |

*Bitwise NOT* inverts the bits of said number from **0** to **1** and vice-versa.

| | | |
|---|---|---|
| `PRINT %!15` | returns **65520** as **15** | (0000 0000 0000 1111) |
| | gets inverted | |
| | to become **65520** | (1111 1111 1111 0000) |
| `PRINT %!43690` | returns **21845** as **43690** | (1010 1010 1010 1010) |
| | gets inverted | |
| | to become **21845** | (0101 0101 0101 0101) |

this seems pretty straightforward correct? Wrong because look at what happens once you omit the **%** symbol and the expression is no longer an integer one:

| | | |
|---|---|---|
| `PRINT !43690` | returns **-43691** | (1010 1010 1010 1010) |
| | as **43690** gets inverted | |
| | **together** with its *sign bit* | |
| | to become **-43691** | (0101 0101 0101 0101) |

what happened can be found in a simple phrase: *two's complement* which we'll examine further below. Until then however run the following examples:

```
10 PRINT !32767
20 PRINT %! 32767
30 PRINT % SGN(!32767)
```

if you feel a bit overwhelmed, not to worry; soon all will be very clear!

## Bitwise operators <<, >>, &, |, ↑, ↑|

*NextBASIC*, can also perform 5 *bitwise operations* (that is operations on the individual binary digits that make up a number) on variables and expressions. These are:

| | |
|---|---|
| *x* << *y* | Shift each bit of *x*, *y* places left |
| *x* >> *y* | Shift each bit of *x*, *y* places right |
| *x* & *y* | Bitwise AND between *x* and *y* |
| *x* \| *y* | Bitwise OR between *x* and *y* |
| *x* ↑ *y* | Bitwise XOR between *x* and *y* *(Deprecated, use the next one)* |
| *x* ↑\| *y* | Bitwise XOR between *x* and *y* (Synonym with the previous) |

More information on Bitwise operations together with examples (as binary examples are much easier to understand) can be found in *Integer Expressions* below. The operators however work on both (except for the deprecated one) Integer and Floating Point expressions

## Logical operators

Standard logical operators can be used within integer expressions if prefixed by a **%**. These are used in the same manner as their floating point counterparts.

| | |
|---|---|
| *x* **AND** *y* | Logical *AND* (gives 0 if *y* is zero, *x* if *y* is non-zero) |
| *x* **OR** *y* | Logical *OR* (gives *x* if *y* is zero, 1 if *y* is non-zero) |
| **NOT** *n* | Logical *NOT* (zero -> 1, non-zero -> 0) |

### Relational operators <, >, = ,<=, >=, <>

| | |
|---|---|
| < | less than |
| > | greater than |
| = | equal to |
| <= | less than or equal to |
| >= | greater than or equal to |
| <> | not equal to |

Whereas you use them in their integer or floating point guise, the six relational operators, work very much in the same way. Like their floating point counterpart, they too produce a result of **0** for *false* and 1 for *true*.

### Expressions

Expressions are useful because, whenever the computer is expecting a number from you, you can give it an expression instead and it will work out the answer. The exceptions to this rule are so few that they will be stated explicitly in every case.

You can add together as many strings (or string variables) as you like in a single expression, and if you want, you can even use parentheses. In the case of Integer Expressions there are some further considerations and limitations as well as additional capabilities (ie. Bitwise operations and modulus) so they warrant a separate examination below.

### Variable names and limitations

We really ought to tell you what you can and cannot use as the names of variables. As we have already said in *Chapter 1,* the name of a string variable has to be followed by **$**; otherwise they can be treated in the same way naming wise: They can use any letters or digits as long as the first one is a letter. You can put spaces in as well to make it easier to read, but they won't count as part of the name. Also, it doesn't make any difference to the name whether you type it in capitals or lowercase letters. There are some restrictions about variable names which are the same as commands (keywords), however, in general, if the variable contains a *NextBASIC* keyword in it (with spaces either side) then it won't be accepted.

Integer variables are a bit different as they can only be a single letter **A** to **Z** (or lower case **a** to **z**) and they're assigned in an expression that begins with a **%** eg:

```
%a=10
```

Additionally, all integer values are treated by default as unsigned 16-bit values except when you use the special **SGN {...}** keyword (in which case they're signed 16-bit – see the relevant section at the end of this chapter for details).

All operations are performed within the confines of 16 bits, meaning all results are truncated to a max value of **65535**, with no checks for overflow/underflow (except division by zero, which results in error **6, Number too big**). Integer variables are pre-allocated and stored in a fixed location outside the normal memory used by *NextBASIC*. This gives a significant speed advantage as well as memory savings compared to the use of ordinary numeric variables.

Further of note is that if a line contains an integer expression, *ALL* variables and arrays contained within the same expression are integer ones. In cases where there is more than one integer expression within a line, each needs to be preceded with a **%**.

Here are some examples of the names of variables that are allowed:

```
x
t42
ItIsWithAHeavyHeartThatIMustSay
nowWeAreSix
```

| | |
|---|---|
| nOWWeaReSiX | (these last two names are considered the same, and refer to the same variable) |

The following are *not* allowed to be the names of variables:

| | |
|---|---|
| pi | PI is a keyword |
| 2001 | (it begins with a digit) |
| A new variable | (contains the separated keyword NEW) |
| 3 bears | (begins with a digit) |
| M*A*S*H | (* is not a letter nor a digit) |
| Fotherington-Thomas | (- is not a letter nor a digit) |

Integer variables can only use the letters A to Z (again, case does not matter, so a to z are also acceptable) – as you can see below, for a variable to be treated as integer, a % symbol somewhere in the same expression must precede it.

## Scientific notation

Numerical expressions can be represented by a number and exponent. Try the following to prove the point:

```
PRINT 2.34e0
PRINT 2.34e1
PRINT 2.34e2
```

and so on up to:

```
PRINT 2.34e15
```

You will see that after a while the computer also starts using scientific notation. Similarly, try:

```
PRINT 2.34e-1
PRINT 2.34e-2
```

and so on.

PRINT gives only eight significant digits of a number. Try:

```
PRINT 4294967295,4294967295-429e7
```

This proves that the computer can hold the digits of 4294967295, even though it is not prepared to display them all at once.

The ZX Spectrum Next, unless integer variables are expressly used (see above), uses floating point arithmetic, which means that it keeps separate the digits of a number (its mantissa) and the position of the point (the exponent). This is not always exact, even for whole numbers.

Type:

```
PRINT 1e10+1-1e10,1e10-1e10+1
```

Numbers are held to about nine and a half digits accuracy, so 1e10 is too big to be held exactly right. The inaccuracy (actually about 2) is more than 1, so the numbers 1e10 and 1e10+1 appear to the computer to be equal. For an even more peculiar example, type:

```
PRINT 5e9+1-5e9
```

Here the inaccuracy in 5e9 is only about 1, and the 1 to be added on in fact gets rounded up to 2. The numbers 5e9+1 and 5e9+2 appear to the computer to be equal.

The largest integer (whole number) that can be held completely accurately is 1 less than 32 2s multiplied together (or 4,294,967,295) – in other words: $2^{32}-1$

The string "" with no characters at all is called the *empty* or *null* string. Remember that spaces are significant and an empty string is not the same as one containing nothing but spaces. Try:

```
PRINT "Have you finished "Finne
gans Wake" yet?"
```

When you press ENTER, you will get the flashing red cursor mark that shows there is a mistake somewhere in the line. When the computer finds the double quotes at the beginning of "Finnegans Wake", it imagines that these mark the end of the string "Have you finished ", and it then can't work out what Finnegans Wake means.

There is a special device to get over this; whenever you want to write a string quote symbol in the middle of a string, you must write it twice, like this:

```
PRINT "Have you finished ""Finn
egans Wake"" yet?"
```

As you can see from what is printed on the screen, each double quote is only really there once; you just have to type it twice to get it recognised.

## Decimal, Binary and Hexadecimal numbers

Number literals in *NextBASIC* can be expressed in *Decimal* (default), *Binary* (preceded by @) and *Hexadecimal* (preceded by $). In the case of integer only literals, the same rule as any with other integer expression applies to binary and hexadecimal literals; they need to be preceded by %, *once* per expression. Consider these examples:

```
PRINT %$E3, @11100011

PRINT $E3, %@11100011

PRINT %$E3+@11100011

PRINT $E3+%@11100011
```

The first example prints an integer and then a floating point number autoconverted from hexadecimal and binary respectively. The same thing happens in the second case but with the first value being a floating point one and the second an integer as again there are two separate expressions following the PRINT keyword. The third example is also valid as it contains a properly marked (preceded by %) integer expression. The addition of the hexadecimal and binary numbers is a single integer expression as the % preceding the addition marked both numbers as integer (and therefore it doesn't need a second %). The fourth example however is NOT valid as it's trying to add a floating point number with an integer number and this expression will fail.

In the case of floating point literals, both hexadecimal and binary numbers can have fractional parts. For example:

| | |
|---|---|
| `BIN 1.1` | is the same as 1.5 (decimal) |
| `@10.01` | is the same as 2.25 (decimal) |
| `$64.c` | is the same as 100.75 (decimal) |

## More about Integer Expressions and Variables

As previously mentioned, the main two reasons for the use of Integer Variables, Arrays and Expressions, is memory efficiency and speed of execution.

Integer variables can be used in assignments (using keywords INPUT, LET, READ, FOR, ENDPROC and PROC) by preceding their name with a % symbol.

Normally, it is not possible to access standard numeric variables or functions within an integer expression, or to access integer variables or operations within a standard numeric expression. In the following program:

```
 10  a=3
 20  b=4
 30  %a=2
 40  %b=5
 50  c=%a*b
 60  d=%b  *  a
 70  PRINT  c,d
 80  %b=b
 90  c=%a*b
100  PRINT  c,d
```

you might expect line 70 to produce 8 and 15. Instead it returns 10 and 10 as the % in lines 50 and 60 indicates that the entire expression is an integer expression, and all the variables named in each line, are integer variables even though each name is not directly preceded by a % and only line 100 produces a different output; 8 and 10 respectively.

It is, as apparent from the above example, possible therefore, to assign an integer expression to a standard normal numeric variable, or vice-versa, and the value will be converted appropriately. This automatic conversion is called *casting* and it's best illustrated in line 80 above as well as the examples below which are all valid assignments:

```
%A=2*PI*radius
```

assigns a truncated floating point calculation to integer variable A

```
%B=%B+(A(7)<<3)
```

shifts integer array element A(7) *left* 3 bits and adds it to integer variable B

```
addr=%x(1)<<8+x(0)
```

calculates standard numeric variable addr from *low* and *high bytes* in integer array X elements 0 and 1.

As we saw earlier it's not *normally* possible to use a floating point expression within an integer expression. But what if we needed to do so? Consider the following example:

```
%a,%b,%c=1:PRINT %a+PI+b+c
```

Looks simple enough, doesn't it? All we expect to happen is for casting to take over and use just the integer portion of the value of PI, but it doesn't work that way. Instead the cursor flashes next to PI and the *NextBASIC* editor complains. To address this, *NextBASIC* includes the special INT {*fp_expression*} keyword (do not omit the braces) which converts (casts) any floating point expression *fp_expression* into an integer. So even if the example above wouldn't work, a small change:

```
%a,%b,%c=1:PRINT %a+INT { PI }+b+c
```

and it works happily! As a matter of fact INT {...} will convert any expression that produces a floating point value. Here are some examples:

```
test = 3.45: PRINT % INT {test}
```

```
alpha = 0: beta = 1: %a = %@0111 + INT
{alpha OR beta)}

%x=%x+INT{(INKEY$="P" OR
INKEY$="p")}-INT{(INKEY$="O" OR
INKEY$="o")}
```

Despite the presence of INT {...}, in order to avoid confusion and unexpected results that can make *debugging*[1] very hard, it would be a good practice to not use one or more single letter standard variables when there's a possibility of a similarly named variable existing in its integer form and instead use a more easily identifiable name.

We discussed about using floating point literals and/or expressions within the integer expressions evaluator; what happens when we want to do the opposite, to use in other words an integer expression as a sub-expression within the standard expression evaluator?

As it happens, this is possible as long as it is started after any opening parenthesis or separating comma. For example:

```
x$=STR$(%a, %b, 5)
x=apples*pears+(%x(3))
```

There is a notable exception to that requirement. For the new BANK... functions (See *Chapter 23* for details about BANK) in the standard expression evaluator, the *bank* number can be considered to be implicitly within parentheses, so it may be specified using an integer expression directly as follows:

```
x=2*PI*BANK %b PEEK myaddress
```

As we saw from the unary ! operator, bitwise operations on variables and arrays are pretty straightforward and involve manipulations of the individual bits of any number as represented in the ZX Spectrum Next's memory.

Shifting left or right involves moving the binary content of a variable x places (bits) to the left or right, padding from the right or left respectively with as many 0s as the places we shift the number for.

To illustrate bit shifting we can do the following example: Let's assign the decimal number 1201 first to an integer variable A, then manipulate its bits by shifting them left and right and printing the result so we can compare:

```
100 %A=1201
110 %A>>=3
120 %A<<=3
130 PRINT %A
```

This will return 1200 when run. To demonstrate what went on we could illustrate the expressions in two consecutive PRINT statements:

```
100 PRINT %1201>>3
110 PRINT %150<<3
```

1 *Debugging is the programming process where you first attempt to ascertain if a program has errors, then to identify these errors and finally to remove them.*

Once we see how the numbers are stored in memory as a series of bits we can easily understand what happened:

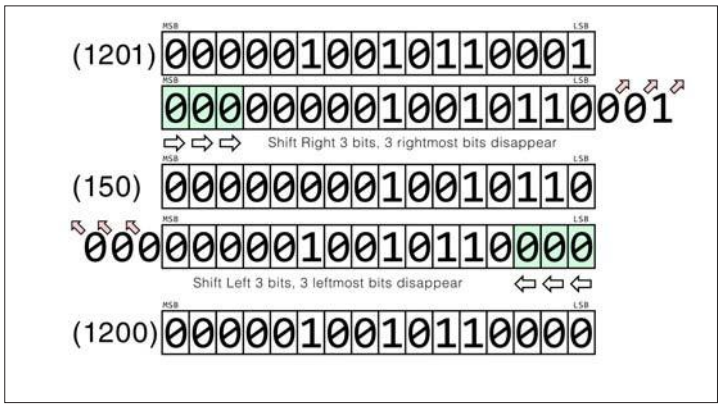



*Fig. 4 - Bit shifting*

What happens however if we do the same to a floating point variable? Let's rewrite the above example:

```
100 A=1201
110 A>>=3
120 A<<=3
130 PRINT A
```

which prints the same number as what we have assigned in line 100! Why the difference? We will need to recruit a *function* from a little further down this manual to help us better understand. Just type the following:

```
100 A=1201: PRINT
    A,STR$(A,2,4)
110 A>>=3: PRINT A,STR$(A,2,4)
120 A<<=3:PRINT A,STR$(A,2,4)
```

The whole trick is in the fractional part of the number we don't normally see because it's 0. By shifting 3 places to the right, we occupied 3 of the fractional places and therefore when we shifted back to the left the number wasn't truncated from the right side of its binary representation!

The remaining bitwise operations are very straightforward. Bitwise AND (&) is used to quickly determine if a bit inside a number is set to 1 or not. The first operand is the number we want to check and the second one is called the *bitmask* which is the number we check against. Consider these two examples:

```
PRINT %@10101010 & @01010101
PRINT @11100011 & @10
```

First example will return 0 while the second 2. The reason for this, is that the numbers in the first example don't have coinciding 1 bits in the same positions while on the second example the second bit will be 1 and as a consequence the bits that match will be the first and second which make binary 10 which in decimal equals 2. To illustrate further:

| | | |
|---|---|---|
| 170 | | 10101010 |
| AND 85 | (Bitmask) | 01010101 |
| Result | | 00000000 |

As you can see, no bit set to 1 in any position of the two numbers matches each other, therefore the result returned is 0 whereas in the second example:

| | | |
|---|---|---|
| 227 | | 11100011 |
| AND 2 | (Bitmask) | 00000010 |
| Result | | 00000010 |

Bit 2 of the mask, matches bit 2 of the number and it is 1 therefore 10 is returned (binary equivalent of decimal 2)

Bitwise OR (|) will return 1 in any position if at least one bit of the two numbers is the same position is 1 and 0 if both are set to 0. For example:

```
PRINT %@10101010 | @01101011
```

will return 235 as only bits in positions 3 and 5 in both numbers are set to 0 making the resulting number 11101011 in binary form (or 235 in decimal). To better illustrate:

| | | |
|---|---|---|
| 170 | | 10101010 |
| OR 107 | (Bitmask) | 01101011 |
| Result | | 11101011 |

Finally, bitwise XOR (↑|) will only return 1 in any position if either bit is set to 1 but *not both*. So two 0s *and* two 1s, both return 0 in a position. Using the same numbers as in the previous example:

```
PRINT %@10101010 ↑| @01101011
```

will return 193 or binary 1100001 since:

| | | |
|---|---|---|
| 170 | | 10101010 |
| XOR 107 | (Bitmask) | 01101011 |
| Result | | 11000001 |

Bitwise expressions are uniquely helpful in determining the condition of flags in several of the ZX Spectrum Next ports (as we will see in *Chapter 22*), since these take the form of individual bits in a binary number and testing those with regular arithmetic can be cumbersome and slow.

## Signed vs Unsigned Integer Expressions

As you saw in *Chapter 1* and in the introduction to this chapter, integer variables in *NextBASIC* are fixed to 16-bits wide *unsigned*, which means that they can display *only positive* integers from 0 to 65535. To illustrate approximately what that means, try the following:

```
PRINT %-64448
```

The computer will respond with: 1088. Keep this result in mind for a moment and then try:

```
PRINT %-32448
```

This time the computer will display the number 33088 on screen. Are you confused yet? Maybe seeing the numbers in binary will help. Let's start with the first response of 1088 and we'll work backwards.

| Decimal | Binary | |
|---|---|---|
| 1088 | 0000 0100 0100 0000 | |
| -64448 | 1 0000 0100 0100 0000 | |
| 64447 | 1111 1011 1011 1111 | Don't mind this for now! |

Aha! Let's now see the second response:

| Decimal | Binary | |
|---|---|---|
| 33088 | 1000 0001 0100 0000 | |
| -32448 | 1000 0001 0100 0000 | |
| 32447 | 0111 1110 1011 1111 | Don't mind this for now! |

Do you now see the pattern? Let's do one more thing that will illustrate how the computer stores the data internally (We'll now jump a bit ahead and borrow a bit from *Chapter 24*).

Type the following program:

```
10 DPOKE 30000, %-32448
20 PRINT % DPEEK 30000: PRINT
   PEEK 30000, PEEK 30001
```

Line 10 enters the entire 16 bits of the value -32448 into memory locations 30000 and 30001, while line 20 first prints what's stored in locations 30000 and 30001 as an unsigned integer and then the individual bytes that make up that value. You will get:

```
33088
64          129
```

The second line just translates to **0100 0000** and **1000 0001** in binary which if we consider that the smallest portion of the 16 bit number was stored first we can rebuild it as: **(129 x 256) + 64** which equals… **33088**!

Now let's first give some background so we can tie all this information together: A *signed* integer is one with either a plus or minus sign in front indicated by one bit in the beginning of the number. Since we have 16 bits assigned to integers and taking the one bit out for the sign, that would leave us 15 bits to display a number with a sign (whereas this sign is positive or negative). Thus a 16 bit signed integer will be able to display numbers to the range of **-32768 to +32767**. This obviously, also means that unsigned integers can have a value twice as high as signed integers. The most common way to represent signed numbers (and the one *NextBASIC* uses) is to use *two's complement* if you recall from earlier in the chapter which works as follows:

On any given binary number representing a decimal *x*, its *two's complement* is a binary number constituted by the first number with inverted digits from **0** to **1** and vice-versa and then adding **1**. The resulting binary number represents decimal *-x*. For example:

For decimal number **2** (represented in 8 bit binary as **00000010**), **-2** would be **00000010**'s *two's complement*. To calculate it we'd have to invert the digits making it **11111101** and then add **1** which would make the resulting number **11111110**. The very first bit signifies the sign (**0** for *positive* and **1** for *negative*). The benefit of using two's complement is that standard arithmetic works properly and any numbers that exceed the bit-width of the numbers get discarded.

After discussing this, the pattern emerging from the previous examples becomes clear!

What happened in the examples above is that *NextBASIC*, in the first example (as mentioned in the beginning of this chapter) truncated the sign bit as it was located in the 17th bit and left us with only the 16 bit unsigned integers of the negative number which is the same as the 16 bit equivalent of the number we fed it. It then tried to interpret the sign bit but since regular integers are unsigned it just returned the positive integer that's represented by the number. For the computer therefore in both cases, what we fed it and what it printed were the exact same number

A further illustration of the above can be shown by using the *unary not* operator (**!**) which as we discussed earlier in the chapter, inverts the number. Let's see:

```
PRINT %!1088, %!33088
```

The computer returns:

```
64447          32447
```

And if we add these together by doing:

```
PRINT %1088 + 64447, %33088 + 32447
```

we will get in both cases 65535!

Integer arithmetic is extremely fast, so we should have at least a way of representing signed integers in *NextBASIC* for both fast calculations as well as special cases, so *NextBASIC* does provide the way to deal with these numbers with the special SGN {...} keyword. What this does is, to treat any integer expression enclosed within it as a signed integer value (ranging from -32768 to 32767). All expressions enclosed within an SGN {...} block are called *signed integer expressions*. *Signed integer expressions* use all the same operators and functions as standard unsigned ones, but the arithmetic operators (+, -, *, /, MOD) and the relational operators (<, <=, >, >=, =, <>) treat their operands as signed values in the range -32768 to 32767. The other operators and functions can be used within a signed integer expression, but still treat their operands as unsigned.

Based on how *two's complement* works, theoretically you can work with just the *two's complement* numbers (which if regarded as unsigned integers, are also positive integers) but in these cases that would be very cumbersome to have to remember the equivalents instead of the actual number we want to involve in our calculation.

If say we need to do 1 + (-32300) - (-1) what would be easier to implement?

```
PRINT % SGN {1}+ SGN {-32300}- SGN {-1}
```

or

```
PRINT %1+33236-65535
```

There are obvious benefits on usability; and also non obvious benefits such as in the following example:

```
10 %x=0
20 PRINT %(x-1)>0,
   %SGN{(x-1)>0}
```

which will result in:

```
1            0
```

on screen as in unsigned expressions 0-1 equals 65535 (see also the previous example) which is obviously larger than 0 while in signed expressions 0-1 equals -1 which is not larger than 0!

SGN {...}, also affects multiplication, division and MODulo operations. Consider this example (which also contains a pitfall!):

```
10 PRINT %10*-1
20 PRINT %10* SGN {-1}
30 PRINT % SGN {10* SGN {-1}}
40 PRINT % SGN {10*-1}
```

If you RUN this, you will see the following on screen:

```
65526
65526
```

```
-10
-10
```

What happened here is that -1 is as we discussed 65535 for unsigned integers. So on line 10, the computer multiplied 10 * 65535 which resulted to 655350 but as an integer number, this is larger than 16 bits. Then it gets truncated to 16 bits which results into 65526 which is obviously wrong as a result. Moving to line 20 we hit the first pitfall discussed in the opening statement: The result of SGN {-1} which is -1 gets converted into an unsigned integer itself so you end up with the exact same situation as with line 10; a multiplication of 10 with 65535. The pitfall therefore here is that SGN{...} must apply to the entirety of the integer expression, so if there are other non-signed expressions they must be taken into consideration when writing each statement! Line 30 produces finally what we were aiming for, but that also happens with line 40! So both are correct but which is the right way to do it?

The answer to that question lies with what we discussed above regarding the "pitfall" with integer expressions. The *subexpression* SGN {-1} will get evaluated to whatever is in the enclosing expression. So if the enclosing expression is an *unsigned expression*, the result of the *subexpression* will also become converted to *unsigned*; ergo since the entirety of the integer expression of line 30 is a *signed expression*, the *signed subexpression* is unnecessary and *may even delay* execution (especially in very complex calculations). The right way therefore to do it, is the way defined in line 40. Obviously this also applied to our initial example which is best written as:

```
PRINT % SGN {1 -32300- (-1)}
```

which is much neater to write AND read!

## Exercises

1. Using the discussion about the unary ! operator and 16 bit binary numbers, calculate and print on screen the *two's complement* for the signed 32 bit integer: 650323

# Chapter 7 – Strings

## Introduction

As we discussed previously strings are series of characters stored as numbers in memory. Which number represents which character is usually governed by a standard. For *NextBASIC* this standard is a modified version of the *ASCII* standard that apart from characters, also defines *tokens* and *UDGs*. See *Chapters 13* and *23* as well as *Appendix A* in order to better understand how characters, tokens and UDGs are stored. This chapter deals with ways of manipulating strings and reiterates some functions touched upon by previous chapters.

## String slicing, using TO

Given a string, a substring of it consists of some consecutive characters from it, taken in sequence. Thus “string” is a substring of “bigger string”, but “b sting” and “big reg” are not.

There is a notation called *slicing* for describing substrings, and this can be applied to arbitrary string expressions. The general form is:

string expression (*start* TO *finish*)

so that, for instance:

```
"abcdef"(2 TO 5)="bcde"
```

If you omit the start, then 1 is assumed; if you omit the finish then the length of the string is assumed. Thus:

```
"abcdef"( TO 5)="abcdef"(1 TO 5)="abcde"
"abcdef"(2 TO )="abcdef"(2 TO 6)="bcdef"
"abcdef"( TO )="abcdef"(1 TO 6)="abcdef"
```

(You can also write this last one as "abcdef"(), for what it's worth.)

A slightly different form misses out the TO and just has one number:

```
"abcdef"(3)="abcdef"(3 TO 3)="c"
```

Although normally both start and finish must refer to existing parts of the string, this rule is overridden by another one: if the start is more than the finish, then the result is the empty string. So:

```
"abcdef"(5 TO 7)
```

gives error 3 Subscript wrong because the string only contains 6 characters and 7 is too many, but:

```
"abcdef"(8 TO 7)="" (an empty string)
```

and:

```
"abcdef"(1 TO 0)="" (again, an empty string)
```

The start and finish must not be negative, or you get error B integer out of range. This next program is a simple one illustrating some of these rules.

```
10 a$="abcdef"
20 FOR n=1 TO 6
30   PRINT a$(n TO 6)
40 NEXT n
```

```
50 STOP
```

Type NEW when this program has been run and enter the next program:

```
10 a$="ABLE WAS I"
20 FOR n=1 TO 10
30 PRINT a$(n TO
   10),a$((11-n) TO 10)
40 NEXT n
```

For string variables, we can not only extract substrings, but also assign to them. For instance, type:

```
a$="I'm the ZX Spectrum Next"
```

and then:

```
a$(5 TO 8)="******"
```

and:

```
PRINT a$
```

Notice how since the substring a$(5 TO 8) is only 4 characters long, only the first four stars have been used. This is a characteristic of assigning to substrings: the substring has to be exactly the same length afterwards as it was before. To make sure this happens, the string that is being assigned to it is cut off on the right if it is too long, or filled out with spaces if it is too short – this is called Procrustean assignment after the road bandit Procrustes who used to make sure that his victims fitted the bed by either stretching them out on a rack or cutting their feet off.

If you now try:

```
a$()="Hello there"
```

and:

```
a$;"."
```

You will see that the same thing has happened again (this time with spaces put in) because a$() counts as a substring.

```
a$="Hello there"
```

will do it properly.

Complicated string expressions will need parentheses around them before they can be sliced. For example:

```
"abc"+"def"(1 TO 2)="abcde"
("abc"+"def")(1 TO 2)="ab"
```

## String multiplication using the * operator

We saw earlier that the multiply (*) operator can be used for string replication. It's syntax is: *a$*n* (takes a string *a$* as the first operand and a number *n* as the second operand, returning a string). If *n* has a fractional part then the string is replicated up to the character that represents the closest integer number to the product of *n times the length* of the original string. In other words, if you multiply a 7 character string by 1.5 times you will get the whole string and an additional 3 characters of it since 7 * 1.5 = 10.5. Finally, the sign of the number determines whether the result is *mirrored* or not. Consider the following examples:

| | |
|---|---|
| `"abcdefg"*2` | returns **abcdefgabcdefg** (two times the string) |
| `"abcdefg"*-1` | returns **gfedcba** (the string is inversed) |
| `"abcdefg"*1.5` | returns **abcdefgabc** |

## Transforming a string with trailing modifers

If you follow a string expression a$ by brackets containing a modifier list in the format:

*a$* [*modifierlist*]

you can transform the string in several useful ways. The *modifierlist* can be any of the following characters:

| | |
|---|---|
| + | convert lower case letters to upper case |
| - | convert upper case letters to lower case |
| < | strip leading spaces (and control characters) |
| > | strip trailing spaces (and control characters) |
| ~ | strip bit 7 (More about bit 7 in *Chapter 23*) terminator from last character of string |
| ^ | add bit 7 terminator to last character of string |
| (f$,r$) | replace any occurrences of characters present in **f$** with the corresponding character from **r$** (or delete if there is no corresponding character) |

The order of the modifiers is unimportant, except for **(f$,r$)** which, if present, must be the final modifier.

The following examples demonstrate what can ne achieved:

```
"    Hello There!    "[<+->]"
```

gives: **hELLO tHERE!**

as trailing and leading spaces are stripped via **<** and **>**and all lower case letters are converted to upper case with the **+** and upper case to lower case with the **-**.

```
"My typewriter is broken"[("nore","dro")]
```

returns **My typwoito is borkd** as *r$* doesn't have a corresponding letter for the **e** of *f$* so all of them are deleted, all **o** are replaced by **r** and all **r** are replaced by **o**

Note that it is possible to slice a modified string, or modify a sliced string, since both **()** and **[]** continue to be evaluated following a string argument until there are no further opening parentheses or brackets. For example:

```
a$(5)[-](3 TO 7)[<]
```

is perfectly valid.

## Tokenisation of strings

If need be, (for example to easily prepare strings to be passed to *functions* **VAL** or **VAL$** – See next *Chapter*) we can tokenise, that is to convert *NextBASIC* reserved words to their single code equivalents (tokens) – without syntax checking– the contents of a string expression *a$* by enclosing it in *braces* **{}** like so:

```
{a$}
```

For example:

| | |
|---|---|
| `{"sin (pi/4)"}` | returns a string "**SIN (PI/4)**", including the tokens **SIN** (code **178**) and **PI** (code **167**) |
| `VAL{"sin (pi/4)"}` | gives **0.7071** |

# Chapter 8 – Functions

Consider a sausage machine. You feed it meat in at one end, turn a handle, and out comes a sausage at the other end. Providing pork meat gives us a pork sausage, beef, a beef sausage and so on.

*Function*s are practically indistinguishable from sausage machines but there is a difference: they work on data instead of meat. You supply one value (called the *argument*), mince it up by doing some calculations or transformations on it, and eventually get another value, the *result*.

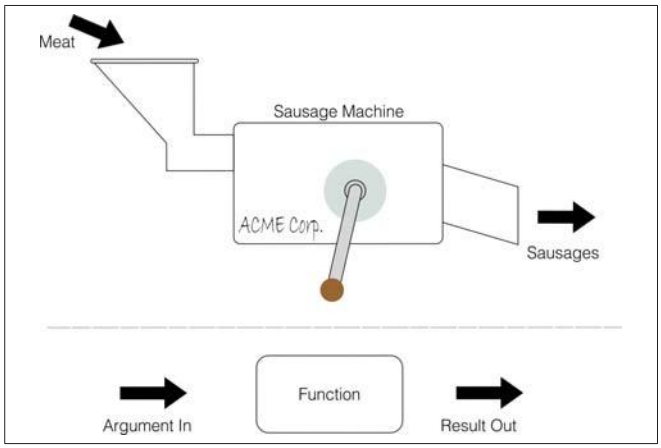



*Fig. 5 – How functions work*

Different arguments give different results, and if the argument is completely inappropriate the function will stop and give an error report.

Just as you can have different machines to make different products – one for sausages. another for dish cloths, and a third for fish-fingers and so on, different functions will do different calculations. Each will have its own value to distinguish it from the others.

You use a function in expressions by typing its name followed by the argument, and when the expression is evaluated the result of the function will be worked out.

Functions, always produce a specific datatype as result and as such they are used in the same way as variables of a certain datatype in expressions In fact string producing functions names end with a $ sign just like regular strings in order to not confuse us. For example running the following (see below for more regarding VAL):

```
10 avalue$="10"
20 b=10+VAL(avalue$)
30 PRINT b
```

will produce 20 which is obviously a number. As far as expressions and *NextBASIC* are concerned VAL (x$) can be used in lieu of any numeric variable or indeed number.

## Order of calculations using functions

Expanding on what we saw on *Chapter 6*, if you mix functions and calculations in a single expression, then the functions' resulting values will be calculated first out before the rest of the operations. Again, however, you can circumvent this rule by using parentheses.

For instance, at the example below, there are two expressions which differ only in the parentheses, and yet the calculations are performed in an entirely different order in each case (although, as it happens, the end results are the same).

Each column shows the succession of operations according to their initial syntax which is found at the top of each column:

```
LEN "Fred"+ LEN "Bloggs"      LEN ("Fred"+"Bloggs")
4+LEN "Bloggs"                LEN ("FredBloggs")
4+6                           LEN "FredBloggs"
10                            10
```

Functions can be separated in *built-in* (contained within *NextBASIC*) and *user-defined* ones (the ones we write). Below we'll visit the *in-built* functions categorised according to the datatype we give them as input to manipulate and/or transform.

Please note that as the functions devoted to maths are a bit more complicated we will list them separately in their own chapter.

## String functions

### LEN

LEN *x$*, works out the length of a string. Its single argument *x$* is the string whose length you want to find, and its result is the length, so that if you type

```
PRINT LEN "ZX Spectrum Next"
```

the answer 16 will be printed on screen, that is the number of characters in *ZX Spectrum Next* (spaces are counted as a character).

### STR$

STR$ converts numbers into strings; that however is not its only capability and for that reason it has two forms. Apart from the simplest task of making a string out of a number in the way it would appear on screen displayed by a PRINT statement (that also means converted – or *cast* – into a decimal number), it can also do the same but converting at the same time the number to a different base!

The simplest form STR$ *x* doen't use parentheses to contain its single argument *x* –a number–, and its result is the string that would appear on the screen if the number were displayed by a PRINT statement. Note how its name ends in a $ sign to show that its result is a string. For example, you could say:

```
a$=STR$ 1e2
```

which would have exactly the same effect as typing:

```
a$="100"
```

since the numeric argument gets converted from scientific notation, to a standard decimal prior to printing. Or you could say:

```
PRINT LEN STR$ 100.000
```

and get the answer 3, because STR$ 100.0000="100".

STR$(*x*, [*base* [, *places*]]) on the other hand uses parentheses, but takes up to 3 numeric arguments. It returns a string representation of number *n* in the optionally specified *base* (2 to 36). For bases > 10, digits larger than 9 are represented with capital letters starting with A. If the base is not specified it's assumed it's 10. If optional argument places is present, then a fractional part of places' digits is also output. Let's first rewrite the example above in a couple of ways:

```
a$=STR$ (1e2)
a$=STR$ (1e2,16)
```

If you assumed prior to trying, that the first produces the same result as the STR$ without parentheses further above, you'd be correct. It too, returns 100. The second however demonstrates the power of conversion as we told STR$ to convert to *base-16* (*hexadecimal*). This returns 64 which is 100 expressed in the hexadecimal system. The following two

examples show what happens when we request both a conversion and the optional places:

```
STR$(201.5, 16)
```

which gives us C9 (which equals 201, the integer part of 201.5 ) and

```
STR$(3.5, 2, 4)
```

which converts 3.5 into a binary number with 4 fractional places for .5 and returns 11.1000.

## IN

IN (*source$, match$ [, startpos[, wild$]]*) returns the leftmost character position number where *match$* was found in *source$*. Optional parameter *startpos* determines the position within *source$* to begin the search (default=1). If *startpos* is negative, the search starts from position ABS(*startpos*) and proceeds backwards (More about ABS further below).

The value returned will be between 1 and LEN *source$* if a match was found or 0 if a match is not found, *startpos* is 0, *match$* or *source$* are the empty string ("").

If you're not entirely sure of the spelling of the word you're looking for within the original string you have the option to use a *wildcard*[1] character which itself is user-definable. *Any character you enter in wild$* or the copyright symbol © (ASCII 127), will become the wildcard character which you can then substitute in *match$* for any character you're unsure about. Any characters in *match$* which are the wildcard character will match any character in *source*$. If *wild$* is the empty string (""), the wildcard character is ASCII 0. Here are some examples to better illustrate:

```
PRINT IN("Harry is awesome"," ")
```

Prints 6 on the screen as a space is a valid character while

```
PRINT IN("Harry is awesome","
       ",7)
```

Prints 9 as we told IN to start looking from position 7 onwards.

```
PRINT IN("Harry is awesome","
       ",-16)
```

will also return 9 as it's looking backwards and finally

```
PRINT IN("Here's
       Garry!","a%%y",1,"%")
```

prints 9 as we substituted any letter for the symbol % and the first string is located there. If however we modified the above slightly and made it:

```
PRINT IN("Here's
       Garry!","%r",1,"%")
```

we'd get 2 as the first match er for the %r is located there. If we then modified it a little further to:

```
PRINT IN("Here's
       Garry!","%r",5,"%")
```

it would have returned 10 as starting from position 5 the first match is the rr in Garry.

1 *A wildcard character is a character placeholder for any possible character*

## VAL and VAL$

VAL *x$* is like STR$ in reverse as it converts string *x$* into the numeric representation of said string. For instance:

```
VAL "3.5"
```

returns 3.5. That is because if you take any number, apply STR$ to it, and then apply VAL to it, you will get back to the number you first thought of. That being said, if you take a string, apply VAL to it, and then apply STR$ to it, you do not always get back to your original string.

VAL is an extremely powerful function, because the string which is its argument is not restricted to looking like a plain number – it can be any numeric expression. Thus, for instance:

```
VAL "2*3"
```

will return 6 or even:

```
VAL ("2"+"*3")
```

will return the same. There are two things happening here. In the first, the argument of VAL is evaluated as a string: the string expression "2"+"*3" is evaluated to give the string "2*3". Then, the string has its double quotes stripped off, and what is left is evaluated as a number; so 2*3 is evaluated to give the number 6.

Now the following can get pretty confusing pretty fast if you do not pay the requisite attention: Remember that inside a string a string quote must be written twice. If you go down into further depths of strings, then you find that string quotes need to be quadrupled or even octupled.

There is another function, rather similar to VAL, called VAL$ *x$*. Its argument *x$* is still a string, but its result is also a string. To see how this works, recall how VAL functions in two steps: first its argument is evaluated as a string, then the double quotes are stripped off this, and whatever is left is evaluated as a number. With VAL$, the first step is the same, but after the string quotes have been stripped off in the second step, whatever is left is evaluated as another string. Thus:

```
VAL$ """Fruit punch"""
```

equals to Fruit Punch (Notice how the string quotes proliferate again.) Do:

```
a$="99"
```

and print out all of the following: VAL a$, VAL "a$", VAL """a$""", VAL$ a$, VAL$ "a$" and VAL$ """a$""". Some of these will work, and some of them won't; try to explain all the answers (Try not to get overly confused).

## Numeric functions

### SGN

SGN *x* is the *sign* function (sometimes called *signum*). It is the first function you have seen that has nothing to do with strings, because both its argument x and its result are numbers. The result is +1 if the argument is positive, 0 if the argument is zero, and -1 if the argument is negative.

## ABS

ABS *x* is another function whose argument *x* and result are both numbers. It converts the argument into a positive number (which is the result) by stripping the sign away, so that for instance:

```
ABS -3.2
```

is the same as:

```
ABS 3.2
```

which in turn equals 3.2

## INT

INT *x* stands for *integer part* – an integer is a whole number, possibly negative. This function converts a fractional number *x* into an integer by throwing away the fractional part, so that for instance:

```
INT 3.9
```

equals 3. Be careful when you are applying it to negative numbers, because it always rounds down: thus, for instance:

```
INT -3.9
```

will return -4

## SQR

SQR calculates the square root of a number – the result that, when multiplied by itself, gives the argument. For instance:

```
SQR 4
```

returns 2 because 2*2 = 4

```
SQR 0.25
```

returns 0.5 because 0.5*0.5 = 0.25 and finally:

```
SQR 2
```

will return 1.4142136 (approximately) because 1.4142136*1.4142136=2.0000001

If you multiply any number (even a negative one) by itself, the answer is always positive. This means that negative numbers do not have square roots, so if you apply SQR to a negative argument you get an error A Invalid Argument.

## User defined functions using DEF and FN

You can also define functions of your own. Their names follow exactly what is valid for procedures. Their arguments/parameters follow the conventions about procedures as well. Additionally just like procedures and subroutines, functions can also be recursive. Recall the factorial example from *Chapter 4* and let's try to express it in a function form

```
DEF FN factor(n)=n?(1,1,n*FN
        factor(n-1))
```

You define a function by putting a **DEF** statement somewhere in the program. For instance, here is the definition of a function **FN s** whose result is the square of the argument:

```
DEF FN sq(x)=x*x: REM square of x
```

The **sq** following the **DEF FN** is the name of the function. The **x** in parentheses is a name by which you wish to refer to the argument of the function.

After the = sign comes the actual definition of the function. This can be any expression, and it can also refer to the argument using the name you've given it (in this case, **x**) as though it were an ordinary variable.

When you have entered this line, you can invoke the function just like one of the computer's own functions, by typing its name, **FN sq**, followed by the argument. Remember that when you have defined a function yourself, the argument must be enclosed in parentheses. Try it out a few times:

```
PRINT FN sq(2)

PRINT FN sq(3+4)

PRINT 1+INT FN sq (LEN "chicken"/2+3)
```

Once you have put the corresponding **DEF** statement into the program, you can use your own functions in expressions just as freely as you can use the computer's.

Note: in some dialects of BASIC you must even enclose the argument of one of the computer's functions in parentheses. This is not the case in *NextBASIC*.

**INT** always rounds down. To round to the nearest integer, add **.5** first – you could write your own function to do this:

```
20 DEF FN r(x)=INT (x+0.5):
   REM gives x rounded to the
   nearest integer.
```

You will then get, for instance:

FN r(2.9) = 3  FN r(2.4) = 2

FN r(-2.9) = -3  FN r(-2.4) = -2

Compare these with the answers you get when you use **INT** instead of **FN r**. Type in and run the following:

```
10 x,y,a=0,0,10
20 DEF FN p(x,y)=a+x*y
30 DEF FN q()=a+x*y
40 PRINT FN p(2,3),FN q()
```

There are a lot of subtle points in this program.

First, a function is not restricted to just one argument: it can have more, or even none at all – but you must still always keep the parentheses.

Second, it doesn't matter whereabouts in the program you put the **DEF FN** statements. After the computer has executed line 10, it simply skips over lines 20 and 30 to get to line 40. They do, however, have to be somewhere in the program. They can't be in a command.

Third, x and y are both the names of variables in the program as a whole, and the names of arguments for the function **FN p**. **FN p** temporarily forgets about the variables called **x** and **y**, but since it has no argument called **a**, it still remembers the variable **a**. In other words the exposed parameters of a function are always **LOCAL** within the function.

Thus when FN p(2,3) is being evaluated, a has the value 10 because it has been initialised in line 10, x has the value 2 because it is the first argument, and y has the value 3 because it is the second argument. Both x and y albeit also defined in line 10 are treated as LOCAL variables and their value doesn't modify their global counterparts. The result is then, 10+2*3=16.

When FN q() is being evaluated, on the other hand, there are no arguments. So a, x and y all still refer to the variables and have values 10, 0 and 0 respectively. The answer in this case is 10+0*0=10.

Now change line 20 to:

```
20 DEF FN p(x,y)=FN q()
```

This time, FN p(2,3) will have the value 10 because FN q will still go back to the variables x and y rather than using the arguments of FN p.

DEF FN can take parameters passed by REFerence as is obvious from the next example:

```
DEF FN ian$(REF
        jen$(),idx)=jen$(idx)
```

Some BASICs (but not *NextBASIC*) have functions called LEFT$, RIGHT$, MID$ and TL$.

LEFT$ (*a$,n*) gives the substring of *a$* consisting of the first *n* characters.

RIGHT$ (*a$,n*) gives the substring of *a$* consisting of the characters from $n^{th}$ on.

MID$ (*a$, $n_1$, $n_2$*) gives the substring of *a$* consisting of $n_2$ characters starting at the $n_1^{th}$.

TL$ (*a$*) gives the substring of *a$* consisting of all its characters except the first.

You can write some user-defined functions to do the same. For example:

```
10 DEF FN TL$(a$)=a$(2 TO )
20 DEF FN LEFT$(a$, n)=a$( TO
     n)
```

Check that these work with strings of length 0 or 1.

Note that our FN LEFT$ has two arguments, one a number and the other a string.

A function *cannot* have integer arguments, nor use integer expressions in its definitions.

## NextBASIC functions within integer expressions

We already discussed the usage of the INT {...} keyword which converts any floating point expression into an integer expression, but in many cases this can be slow. In other cases the values produced by a function are either plain 8 or 16 bit integers which means that integer-only versions of said function would provide significant boost over their standard counterparts. *NextBASIC* caters for these cases with special integer-only forms of the following functions:

| | |
|---|---|
| ABS *n* | Return the ABSolute value of *n* – See this *Chapter* |
| IN *n* | Read value from *Hardware Port n* – See *Chapter 22* |
| INPUT *n* | Read/Define input controllers – See *Chapters 17 and 22* |
| REG *n* | Read value from *Next Register n* – See *Chapter 22* |
| PEEK *a* | Read byte from address *a* in memory – See *Chapter 23* |

| | |
|---|---|
| DPEEK *a* | Read word[2] from memory (double PEEK) – See *Chapter 23* |
| USR[3] *a* | Execute Machine Code routine – See *Chapter 25* |
| USR$ *a* | Execute Machine Code routine – See *Chapter 25* |
| BIN *n* | Synonym for @*n*, specifying binary values |
| RND *n* | Generates pseudo-random value in range 0 to n–1 (equivalent to floating-point INT (RND*n) ) |
| BANK *b* PEEK *o* | Read byte at offset *o* from bank *b* – See *Chapter 23* |
| BANK *b* DPEEK *o* | Read word at offset *o* from bank *b* (double PEEK) – See *Chapter 23* |
| BANK *b* USR *o* | Execute Machine Code routine at offset *o* in bank *b* – See *Chapters 23* and *25* |
| BANK *b* USR$ *o* | Execute Machine Code routine at offset *o* in bank *b* – See *Chapters 23* and *25* |

These are written by including a % sign in front of them like all integer expressions. For example to read from hardware port 254:

```
%a = % IN 254
```

Or to check what speed your ZX Spectrum Next is running (masking the speed bits of NextREG 7) you could give :

```
PRINT %REG 7 & BIN 00000011
```

Randomly read a byte from the ROM:

```
%a=%RND 16384:PRINT %a,% PEEK a
```

## Exercise

1. Use the function FN sq(x)=x*x to test SQR. You should find that:

   FN sq(SQR x)=x

   if you substitute any positive number for x, and:

   SQR FN s(x)=ABS x

   whether x is positive or negative (Why the ABS?)

2. Write functions FN RIGHT$ and FN MID$

2 *A word in standard computer terminology is a two-byte (ie. 16 bit) value. 32 bit values (two-word) are called Long Words.*

3 *USR and USR$ are very special functions as they also act like commands but using a function syntax*

# Chapter 9 – Mathematical Functions

This chapter deals with the mathematics that the *ZX Spectrum Next* can handle. Quite possibly you will never have to use any of this at all, so if you find it too heavy going, don't be afraid of skipping it. It covers the operation ↑ (raising to a power), the functions **EXP** and **LN**, and the trigonometrical functions **SIN**, **COS**, **TAN** and their inverses **ASN**, **ACS**, and **ATN**.

## ↑ and EXP

You can raise one number to the power of another – that means: *multiply the first number by itself the second number of times*. This is normally shown by writing the second number just above and to the right of the first number like so $2^3$; but since this gets unnecessarily complex to write and display on a computer, we use the symbol ↑ instead. For example, the powers of 2 are:

```
2↑1=2
2↑2=2*2=4          (2 squared)
2↑3=2*2*2=8        (2 cubed)
2↑4=2*2*2*2=16     (2 to the fourth power)
```

Thus at its most elementary level, a↑b means *a multiplied by itself b times*, but obviously this only makes sense if *b* is a positive whole number. To find a definition that works for other values of *b*, we consider the rule:

```
a↑(b+c) = a↑b*a↑c
```

(Notice that we give ↑ a higher priority than * and / so that when there are several operations in one expression, the ↑s are evaluated before the *s and /s.) You should not need much convincing that this works when *b* and *c* are both positive whole numbers; but if we decide that we want it to work even when they are not, then we find ourselves compelled to accept that:

| | | |
|---|---|---|
| a↑0 | = | 1 |
| a↑(-b) | = | 1/a↑b |
| a↑(1/b) | = | the $b_{th}$ root of *a*, which is to say, *the number that you have to multiply by itself b times to get a*. |

and:

| | | |
|---|---|---|
| a↑(b*c) | = | (a↑b)↑c |

If you have never seen any of this before then don't try to remember it straight away; just remember that:

| | | |
|---|---|---|
| a↑(-1) | = | 1/a |

and:

| | | |
|---|---|---|
| a↑(1/2) | = | SQR a |

and maybe when you are familiar with these the rest will begin to make sense.

Experiment with all this by trying this program:

```
10 INPUT a,b,c
20 PRINT a↑(b+c),a↑b*a↑c
30 GO TO 10
```

Of course, if the rule we gave earlier is true, then each time round the two numbers that the computer prints out will be equal. (Note – because of the way the computer works out ↑, the number on the left – *a* in this case – must never be negative.)

A rather typical example of what this function can be used for is that of compound interest. Suppose you keep some of your money in a building society and they give 15% interest per year. Then after one year you will have not just the 100% that you had anyway, but also the 15% interest that the building society have given you, making altogether 115% of what you had originally. To put it another way, you have multiplied your sum of money by 1.15, and this is true however much you had there in the first place. After another year, the same will have happened again, so that you will then have 1.15*1.15=1.15↑2=1.3225 times your original sum of money. In general, after y years, you will have 1.15↑y times what you started out with.

If you try this command:

```
FOR y=0 TO 100:PRINT y,10*1.15↑y
:NEXT y
```

you will see that even starting off from just £10, it all mounts up quite quickly, and what is more, it gets faster and faster as time goes on. (Although even so, you might still find that it doesn't keep up with inflation.)

This sort of behaviour, where after a fixed interval of time some quantity multiplies itself by a fixed proportion, is called *exponential growth*, and it is calculated by raising a fixed number to the power of the time. Suppose you did this:

```
10  DEF FN a(x)=a↑x
```

Here, **a** is more or less fixed, by **LET** statements: its value will correspond to the interest rate, which changes only every so often.

There is a certain value for **a** that makes the function **FN a** look especially pretty to the trained eye of a mathematician and this value is called $e$. *NextBASIC* has a function called **EXP** defined by:

EXP x=$e$↑x

Unfortunately, $e$ itself is not an especially pretty number: it is an infinite non-recurring decimal. You can see its first few decimal places by doing:

```
PRINT EXP 1
```

because **EXP 1** = $e$↑1 = $e$. Of course, this is just an approximation. You can never write down $e$ exactly.

## LN

The inverse of an exponential function is a logarithmic function: the *logarithm* (to *base*$_a$) of a number *x* is the power to which you have to raise *a* to get the number *x*, and it is written $log_a x$. Thus by definition a↑$log_a$x=x; and it is also true that $log$(a↑x)=x. You may well already know how to use *base*$_{10}$ *logarithms* for doing multiplications; these are called *common logarithms*. *NextBASIC* has a function **LN** which calculates *logarithms* to the *base*$_e$; these are called *natural logarithms*. To calculate logarithms to any other base, you must divide the *natural logarithm* by the *natural logarithm* of the base:

$log_a x$ = LN x/ LN a

## PI

Given any circle, you can find its perimeter (the distance round its edge; often called its circumference) by multiplying its diameter (width) by a number called π. (π is a Greek p, and it is used because it stands for the Greek word *perimeter*. Unlike, what's commonly believed, its pronunciation is the same as in English.)

Like $e$, $\pi$ is an infinite non-recurring decimal; it starts off as 3.141592653589.... The word PI in *NextBASIC* is taken as standing for this number – try PRINT PI.

## Trigonometry with SIN, COS, TAN, ASN, ACS and ATN

The trigonometrical functions measure what happens when a point moves round a circle. Here is a circle of *radius* 1 (1 what? It doesn't matter, as long as we keep to the same unit all the way through. There is nothing to stop you inventing a new unit of your own for every circle that you happen to be interested in) and a point moving round it. The point started at the *3 o'clock* position, and then moved round in an anti-clockwise direction.

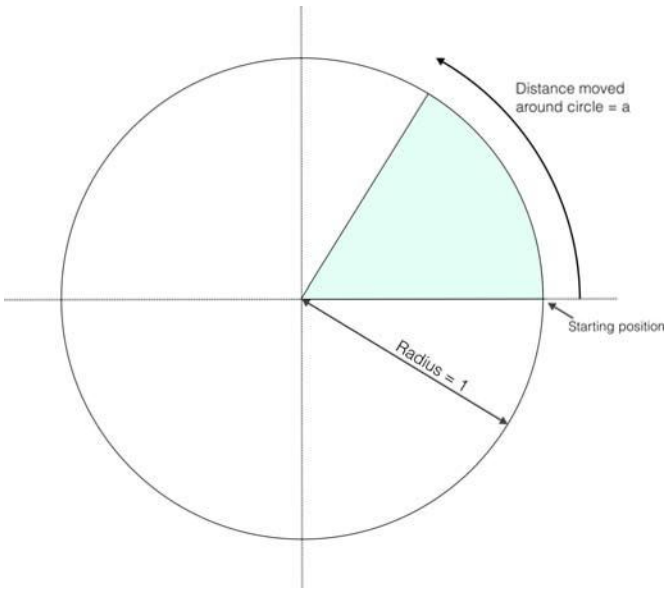



*Fig. 6 – Basics of trigonometrical measurements*

We have also drawn in, two lines called axes through the centre of the circle. The one through *9 o'clock* and *3 o'clock* is called the *x-axis*, and the one through *6 o'clock* and *12 o'clock* is called the *y-axis*. To specify where the point is, you say how far it has moved round the circle from its *3 o'clock* starting position: let us call this distance *a*. We know that the circumference of the circle is $2\pi$ (because its radius is 1 and its diameter is thus 2): so when it has moved a quarter of the way round the circle, $a = \pi/2$; when it has moved halfway round, $a = \pi$; and when it has moved the whole way round, $a = 2\pi$.

Given the curved distance round the edge, *a*, two other distances you might like to know are how far the point is to the right of the *y-axis*, and how far it is above the *x-axis*. These are called, respectively, the *cosine* and *sine* of *a*. The functions COS and SIN on the computer will calculate these.

Note that if the point goes to the left of the *y-axis*, then the *cosine* becomes negative; and if the point goes below the *x-axis*, the *sine* becomes negative.

Another property is that once a has got up to $2\pi$, the point is back where it started and the *sine* and *cosine* start taking the same values all over again:

```
SIN (a+2*PI) = SIN a
COS (a+2*PI) = COS a
```

The *tangent* of *a* is defined to be the *sine* divided by the *cosine*; the corresponding function on the computer is called TAN.

Sometimes we need to work these functions out in reverse, finding the value of *a* that has given *sine, cosine* or *tangent*. The functions to do this are called *arcsine* (ASN on the computer), *arccosine* (ACS) and *arctangent* (ATN).

In the diagram of the point moving round the circle, look at the radius joining the centre to the point. You should be able to see that the distance we have called *a*, the distance

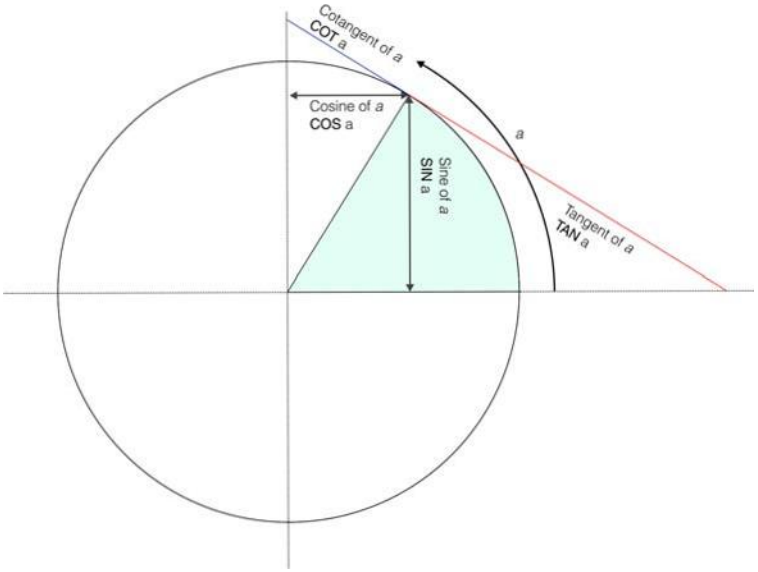



*Fig. 7 – Graphical representation of trigonometrical functions*

that the point has moved round the edge of the circle, is a way of measuring the angle through which the radius has moved away from the x-axis.

When $a=\pi/2$, the angle is **90**° (degrees).
When $a=\pi$, the angle is **180**°; and so round to when $a=2\pi$, and the angle is **360**°.

You might just as well forget about degrees, and measure the angle in terms of *a* alone: we say then that we are measuring the angle in radians. Thus $\pi/2$ radians$=$**90**° and so on.

You must always remember that in *NextBASIC* **SIN**, **COS** and so on use *radians* and not *degrees*. To convert *degrees* to *radians*, divide by **180** and multiply by $\pi$; to convert back from *radians* to *degrees*, you divide by $\pi$ and multiply by **180**.

## Exercises

1. Using the knowledge you have gained from this chapter, define a function to convert radians to degrees (this may prove very useful to you in the future).

2. In *Fig. 7* above, the function **COT** appears while it's not part of *NextBASIC*'s vocabulary. Write a function that returns the value of the *cotangent* of *a* using **TAN**.

# Chapter 10 – Random Numbers

## RANDOMIZE, RND and % RND

This chapter deals with the functions **RND, RND ()** and **% RND** and the keyword **RANDOMIZE**. They are all used in connection with random numbers, so you must be careful not to get them mixed up.

As far as normal functions go, **RND** is quite unusual: although it does calculations and produces a result, it does not need an argument.

Each time you use it, its result is a new *random floating point number* Sbetween **0** and **1**. (Sometimes it can take the value **0**, but never **1**.)

Try:

```
10 PRINT RND
20 GO TO 10
```

to see how the answer varies. Can you detect any pattern? You shouldn't be able to; *random* means that there is no pattern[1].

**% RND**, which is – as seen on *Chapter 8* – the version of **RND** available in integer expressions, behaves slightly differently. It takes a single argument (e.g. *n*) and returns a random integer in the range **0** to **n-1**. For example, **%RND 10** will return a random integer between **0** and **9**.

While **RND** returns, as discussed above, a random number between **0** and **1**, you can easily get random numbers in other ranges. For instance, **5*RND** is between **0** and **5**, and **1.3+0.7*RND** is between **1.3** and **2**. For cases where we need to be in the standard expression evaluator, there is yet another version of **RND**, which is not an integer expression only function:

**RND** (*n*)

which returns a random integer between **0** and **n-1** like **%RND n**. This is recommended over using the standard fractional floating-point function **RND** since it doesn't suffer from the biasing inherent in converting a fractional random number to an integer with multiply and truncation steps.

To get whole numbers with **RND** use **INT** (remembering that **INT** always rounds down) as in **1+INT (RND*6)**. If however your desired random values can stay within the range of **0** to **65535,** it is better to use **% RND or RND()** which avoid the unnecessary conversions – and rather slow – floating point calculations involved.

To illustrate better what all version can do, let's use all three of them in a program to simulate dice throwing. **RND*6** is in the range **0** to **6**, but since it never actually reaches **6, INT (RND*6)** is **0**,**1**,**2**,**3**,**4** or **5**.

Here is the dice throwing program:

```
10 REM dice throwing program
20 CLS
30 FOR n=1 TO 2
40 PRINT 1+INT (RND*6);" ";
50 NEXT n
60 INPUT a$: GO TO 20
```

1 *Actually, RND is not truly random, because it follows a fixed sequence of 65536 numbers. However, these are so thoroughly jumbled up that there are at least no obvious patterns so we say that RND is pseudo-random.*

Press ENTER each time you want to throw the dice. To use % RND instead, change line 40 to read:

```
40 PRINT %1+ RND 6;" ";
```

and to use RND () you need to write line 40 as:

```
40 PRINT 1+ RND (6);" ";
```

Aren't the latter two more readable?

The RANDOMIZE statement, is used to make RND and % RND start off at a definite place in its sequence of numbers, as you can see with this program:

```
10 RANDOMIZE 1
20 FOR n=1 TO 5: PRINT % RND
   100,: NEXT n
30 PRINT: GO TO 10
```

After each execution of RANDOMIZE 1, the % RND sequence starts off again with 97 and if you use RND instead of % RND 100, you'll get 0.0022735596. You can use other numbers between 1 and 65535 in the RANDOMIZE statement to start the RND sequence off at different places.

If you had a program with RND, RND() or %RND in it and it also had some mistakes that you had not found, then it would help to use RANDOMIZE like this so that the program behaved the same way each time you ran it.

RANDOMIZE on its own (and RANDOMIZE 0 has the same effect) is different, because it really does randomise RND, RND() and % RND – you can see this in the next program:

```
10 RANDOMIZE
20 PRINT % RND 65535: GO TO 10
```

The sequence you get here is not very random, because RANDOMIZE uses the time since the computer was switched on. Since this has gone up by the same amount each time RANDOMIZE is executed, the next % RND does more or less the same. You would get better randomness by replacing GO TO 10 by GO TO 20. Here is a program to toss coins and count the numbers of heads and tails.

```
10 heads,tails=0
20 coin=% RND 2
30 ON coin: heads+=1:tails+=1
40 PRINT heads;",";tails,
50 IF tails<>0 THEN PRINT
   heads/tails;
60 PRINT: GO TO 20
```

The ratio of heads to tails should become approximately 1 if you go on long enough, because in the long run you expect approximately equal numbers of heads and tails.

Note that RANDOMIZE can also be written in short as RAND and it will expand to RANDOMIZE!

## Exercises

1. (*For mathematicians only.*)
   Let *p* be a (large) prime, and let *a* be a primitive root *modulo p*.
   Then if $b_i$ is the residue of $a_i$ *modulo p* ($1 \le b_i \le p\text{-}1$ ), the sequence:

   $$\frac{b_i\text{-}1}{p\text{-}1}$$

   is a cyclical sequence of *p*-1 distinct numbers in the range **0** to **1** (excluding **1**).
   By choosing *a* suitably, these can be made to look fairly random.
   **65537** is a Fermat prime, $2^{16}+1$. Because the multiplicative group of non-zero residues *modulo* **65537** has a power of **2** as its order, a residue is a primitive root if and only if it is not a quadratic residue. Use Gauss' law of quadratic reciprocity to show that **75** is a primitive root *modulo* **65537** .
   The *ZX Spectrum Next* uses *p*=**65537** and *a*=**75**, and stores some $b_i$-1 in memory. **RND** entails replacing $b_i$-1 in memory by $b_{i+1}$-1, and yielding the result ($b_{i+1}$-1) / (*p*-1).
   **RANDOMIZE n** (with $1 \le$ **n** $\le$ **65535**) makes $b_i$ equal to $n+1$.
   **RND** is approximately uniformly distributed over the range **0** to **1**.

# Chapter 11 – Arrays

## DIM

Suppose you have a list of numbers, for instance the marks of ten people in a class. To store them in the computer you could set up a single variable for each person, but you would find them very awkward. You might decide to call the variable Bloggs 1, Bloggs 2, and so on up to Bloggs 10, but the program to set up these ten numbers would be rather long and boring to type in.

How much nicer it would be if you could type this:

```
 5 REM this program will not
   work
10 FOR n=1 TO 10
20    READ Bloggs n
30 NEXT n
40 DATA 10,2,5,9,16,3,11,1,0,6
```

Well, you can't!

However, there is a mechanism by which you can apply this idea, and it uses *arrays*. An *array* is a set of variables, its *elements*, all with the same name, and distinguished only by a number (the *subscript*) written in parentheses after the name. In our example the name could be b and the ten variables would then be b(1), b(2), and so on up to b(10).

The *elements* of an *array* are called *subscripted variables*, as opposed to the simple variables that you are already familiar with.

Before you can use an *array*, you must reserve some space for it inside the computer, and you do this using a DIM (for dimension) statement:

```
DIM b(10)
```

sets up an array called b with dimension 10 (i.e. there are 10 *subscripted variables* b(1),...,b(10)) and initialises the 10 values to 0. It also deletes any *array* called b that existed previously. (But not a simple variable. An *array* and a simple numerical variable with the same name can coexist, and there shouldn't be any confusion between them because the *array* variable always has a *subscript*). The *subscript* can be an arbitrary numerical expression, so now you can write:

```
 5 DIM b(10)
10 FOR n=1 TO 10
20    READ b(n)
30 NEXT n
40 DATA 10,2,5,9,16,3,11,1,0,6
```

to read in the elements from a DATA list, or:

```
10 FOR %n=1 TO 10
20 INPUT %m(n)
30 NEXT %n
```

to INPUT the elements' values by hand. Note, that in the second example there is no DIM statement. That's because as discussed in *Chapter 1,* the *second array is an integer array*. *Integer arrays* come *predimensioned* to a fixed 64 elements numbered 0 to 63. Attempting

to enter a DIM statement for %m will produce an audible tone and entering the statement will not be successful.

If we need to use an integer array with more than 64 elements, it is possible although what changes is the way we have to address them. Whereas in a normal integer array the subscript is written inside parentheses () for integer arrays *larger-than-64-elements*, the subscript is written within brackets []. Furthermore, *larger-than-64-elements integer arrays* reduce the number of available integer arrays in the system as they take the entire array that follows sequentially from the one we're using and attach it to the current one. What this means is that if we want to use a 128 element integer array %a[], this will take the space from integer array %b(). If we want to use an 192 element integer array %c[], this will use space from integer arrays %d() and %e() and so on.

The maximum integer array usable is 26 x 64 =1664 if using integer array %a[] with no other arrays available. Note that subsequent arrays don't disappear; they're still accessible carrying data from the integer array that reserved them. Modifying them however may have unexpected consequences. To illustrate this point, let's assume an integer array %a[] with a desired 128 elements. Write the following little program:

```
10 %a[65] = 43
20 PRINT %a[65]
30 PRINT %b(1): REM the 65th
   element of array a[] is
   b(1)
```

It's now obvious how this works!

You can also set up *arrays* with more than one *dimension*. This does also apply to *Integer Arrays,* although they're normally predefined to have a *single* dimension; you'll see how below. In a *two-dimensional array* you need two numbers to specify one of the *elements* – rather like the line and column numbers to specify a character position on the television screen – so it has the form of a table or matrix.

Alternatively, if you imagine the line and column numbers (two *dimensions*) as referring to a printed page, you could have an extra *dimension* for the page numbers. Of course, we are talking about *numeric arrays*; so the elements would not be printed characters as in a book, but numbers. Think of the elements of a *three-dimensional* array v as being specified by v (*page number, line number, column number*).

For example, to set up a *two-dimensional array* c with dimensions 3 and 6, you use a DIM statement:

```
DIM c(3,6)
```

This then gives you 3 x 6=18 *subscripted variables*:

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | c(1,1) | c(1,2) | c(1,3) | c(1,4) | c(1,5) | c(1,6) |
| 2 | c(2,1) | c(2,2) | c(2,3) | c(2,4) | c(2,5) | c(2,6) |
| 3 | c(3,1) | c(3,2) | c(3,3) | c(3,4) | c(3,5) | c(3,6) |

*Table 3 – Representation of a two-dimensional array*

The same principle works for any number of *dimensions*.

Although you can have a number and an *array* with the same name, you *cannot have two arrays with the same name*, even if they have different numbers of *dimensions except* in the case of normal numerical and integer arrays.

As we mentioned above integer arrays can have a second dimension as well. This follows the discussion of extending integer arrays to larger than 64 elements. The technique is similar; If a *two-dimensional* integer array is required, we enclose subscripts within brackets **[]**. The difference here is that subscripts need to be individually enclosed: For example whereas we would address regular array **c()** defined with **DIM c(4,64)** with **c(x,y)** in the case of its integer counterpart we would address it as **%c[x][y]**. Each **x** dimension takes one entire array that follows the base array name. For example using **%c [x][y]** with **x=0** to **5** and y= **0** to **63** will use arrays **%C(),%D(),%E(),%F(),%G() and %H()**

There are also *string arrays*. The strings in an array differ from simple strings in that they are of fixed length and assignment to them is always Procrustean – chopped off or padded with spaces. Another way of thinking of them is as *arrays* (with one extra *dimension*) of *single characters*. The name of a *string array* is a standard variable name followed by **$**, and a *string array* and a simple string variable *cannot* have the same name (unlike the case for numbers).

Suppose then, that you want an *array* **a$** of three strings. You must decide how long these strings are to be – let us suppose that **10** characters each is long enough. You then say:

```
DIM a$(3,10)
```
(type this in)

This sets up a **3*10** *array of characters*, but you can also think of each *row* as being a string:

| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | a$(1) | a$(1,1) | a$(1,2) | a$(1,3) | a$(1,4) | a$(1,5) | a$(1,6) | a$(1,7) | a$(1,8) | a$(1,9) | a$(1,10) |
| 2 | a$(2) | a$(2,1) | a$(2,2) | a$(2,3) | a$(2,4) | a$(2,5) | a$(2,6) | a$(2,7) | a$(2,8) | a$(2,9) | a$(2,10) |
| 3 | a$(3) | a$(3,1) | a$(3,2) | a$(3,3) | a$(3,4) | a$(3,5) | a$(3,6) | a$(3,7) | a$(3,8) | a$(3,9) | a$(3,10) |

*Table 4 – Representation of a string array*

If you give the same number of *subscripts* (two in this case) as there were *dimensions* in the **DIM** statement, then you get a single character; but if you miss the last one out, then you get a *fixed length string*. So, for instance, **a$(2,7)** is the 7th character in the string **a$(2)**; using the slicing notation, we could also write this as **a$(2)(7)**. Now type:

```
a$(2)="1234567890"
```

and:

```
PRINT a$(2),a$(2,7)
```

You get:

```
1234567890          7
```

For the last *subscript* (the one you can miss out), you can also have a slicer, so that for instance:

```
a$(2,4 TO 8) = a$(2)(4 TO 8) = "45678"
```

*Remember*: in a *string array*, all the strings have the same *–fixed–* length. The **DIM** statement has an extra number (the last one) to specify this length. When you write down a *subscripted variable* for a *string array*, you can put in an extra number, or a slicer, to correspond with the extra number in the **DIM** statement. You can have string arrays with no dimensions. Type:

```
DIM a$(10)
```

and you will find that **a$** behaves just like a string variable, except that it always has *length* **10**, and assignment to it is always Procrustean. Whatever part of the value doesn't fit gets left out.

## DIM function

Apart from the DIM array declaration command, there's also a function by the same name that returns information regarding any declared array. It's syntax is:

DIM (*arrayname[$]()* [, *dimension*]) |*numeric* and it returns the number of elements in the specified *dimension* of the array *arrayname* (dimension defaults to 0 if not specified).

If *dimension* equals 0, DIM will simply return the number of dimensions in the array.

Simple strings are treated as single-dimension character arrays, returning 1 as the number of dimensions and the current string length as the number of elements in dimension 1. Let's write a little program to illustrate:

```
10 DIM a(100,10,5)
20 PRINT DIM (a())
30 PRINT DIM(a(),1)
40 PRINT DIM(a(),2)
50 PRINT DIM(a(),3)
```

which will return:

```
  3
100
 10
  5
```

## Exercises

1. Use READ and DATA statements to set up an array m$ of twelve strings in which m$(n) is the name of the *n*th month. (Hint: the DIM statement will be DIM m$(12,9). Test it by printing out all the m$(n) (use a loop)).

2. Type:

```
PRINT "now is the month of
";m$(5);"ing"; " when
merry lads
are playing"
```

   What can you do about all those spaces?

# Chapter 12 – Conditions

## AND, OR and NOT

We saw in *Chapter 2 how an* IF statement takes the form:

IF *condition* ...

Apart from expressions that generate true (1) or false (0) results, the conditions there, were the relations (=, <, >, <=, >= and <>), which compare two numbers or two strings. You can also combine several of these, using the logical operations, AND, OR and NOT.

One relation AND another relation is *true* whenever both relations are *true*, so you could have a line like:

```
IF a$="yes" AND x>0 THEN PRINT x
```

in which x only gets printed if a$="yes" and x>0. The syntax here is so close to English that it hardly seems worth spelling out the details. As in English, you can join lots of relations together with AND, and then the whole lot is *true* if all the individual relations are.

One relation OR another is *true* whenever at least one of the two relations is *true*. (Remember that it is still *true* if both the relations are *true*; this is not always implied in English).

The NOT relationship turns things upside down. The NOT relation is *true* whenever the relation is *false*, and *false* whenever it is *true*!

*Logical expressions*, can be made with relations and AND, OR and NOT, just as numerical expressions can be made with numbers and +, - and so on; you can even put them in parentheses if necessary. They have priorities in the same way as the usual operations +, -, *, / and ↑ do: OR has the lowest priority, then AND, then NOT, then the relations, and the usual operations.

NOT is really a function, with an argument and a result, but its priority is much lower than that of other functions. Therefore its argument does not need parentheses unless it contains AND or OR (or both). NOT a=b means the same as NOT (a=b) (and the same as a<>b, of course).

<> is the negation of = in the sense that it is *true if, and only if*, = is *false*. In other words:

a<>b is the same as NOT a=b

and also:

NOT a<>b is the same as a=b

Persuade yourself that >= and <= are the negations of < and > respectively: thus you can always get rid of NOT from in front of a relation by changing the relation.

Also:

NOT (*a first logical expression* AND *a second*)

is the same as:

NOT (*the first*) OR NOT (*the second*)

and:

NOT (*a first logical expression* OR *a second*)

is the same as:

NOT (*the first*) AND NOT (*the second*)

Using this, you can work NOTs through parentheses until eventually they are all applied to relations, and then you can get rid of them. Logically speaking, NOT is unnecessary, although you might still find that using it makes a program clearer.

The following section is quite complicated, and can be skipped by the fainthearted!

Try:

```
PRINT 1=2,1<>2
```

which you might expect to give a syntax error. In fact, as far as the computer is concerned, there is no such thing as a logical value: instead it uses ordinary numbers, subject to a few rules.

1. =, <, >, <=, >= and <> all give numeric results: 1 for *true*, and 0 for *false*. Thus the PRINT command above printed 0 for 1=2, which is *false*, and 1 for 1<>2, which is *true*.
2. In: IF *condition* THEN … the condition can be actually any numeric expression. If its value is 0, then it counts as *false*, and any other value (including the value of 1 that a *true* relation gives) counts as *true*. Thus the IF statement means exactly the same as:
   IF *condition* <>0 THEN . . .
3. AND, OR and NOT are also number-valued operations.

   x AND *y* has the value
   - x if *y* is *true* (non-zero)
   - 0 (*false*), if *y* is *false* (zero)

   *x* OR *y* has the value
   - 1 (true) if *y* is *true* (non-zero)
   - x, if *y* is *false* (zero)

   NOT *x* has the value
   - 0 (*false*), if *x* is *true* (non-zero)
   - 1 (*true*), if *x* is *false* (zero)

(Notice that *true* means *non-zero* when we're checking a given value, but it means 1 when we're producing a new one.)

Read through the chapter again in the light of this revelation, making sure that it all works.

In the expressions x AND y, x OR y and NOT x, *x* and y will usually take the values 0 and 1 for *false* and *true*. Work out the ten different combinations (four for AND, four for OR and two for NOT) and check that they do what the chapter leads you to expect them to do.

Try this program:

```
10 INPUT a
20 INPUT b
30 PRINT (a AND a>=b)+(b AND
   a<b)
40 GO TO 10
```

Each time it prints the larger of the two numbers a and b.
Convince yourself that you can think of:

x AND y as meaning: *x* if *y* (else the result is 0)

and of:

x OR y as meaning: *x* unless *y* (in which case the result is 1)

An expression using **AND** or **OR** like this is called a *conditional expression*.

An example using **OR** could be:

```
price=price_less_tax*(1.15 OR v$="zero
rated")
```

Notice how **AND** tends to go with addition (because its default value is **0**), and **OR** tends to go with multiplication (because its default value is **1**).

You can also make string valued conditional expressions, but only using **AND**.

**x$ AND y** has the value { **x$** if *y* is non-zero / **""** if *y* is zero

So it means **x$ if y** (else the empty string).

Try this program, which inputs two strings and puts them in alphabetical order:

```
10 INPUT "Type in two
   strings";a$,b$
20 IF a$>b$ THEN
   c$=a$:a$=b$:b$=c$
30 PRINT a$;" ";("<" AND a$
   <b$)+("=" AND a$=b$);
   " ";b$
40 GO TO 10
```

## Exercise

1. *NextBASIC* can sometimes work along different lines from English. Consider, for instance, the English clause *If a doesn't equal b or c*. How would you write this in *NextBASIC*? The answer is not:

   ```
   IF a<>b OR c
   ```

   nor is it

   ```
   IF a<>b OR a<>c
   ```

# Chapter 13 – The Character Set

The letters, digits, punctuation marks and so on that can appear in strings are called characters, and they make up the alphabet, or character set that the ZX Spectrum Next uses.

## CHR$ and CODE

As you will also see in *Appendix A*, there are *256* character locations, and each one is assigned a code between *0* and *255*. To convert between codes and characters, two functions exist: CODE and CHR$. CODE is applied to a string, and returns the code of the first character in the string (or 0 if the string is empty). CHR$ is applied to a code, and returns the single character string that corresponds to that code. We started this paragraph by saying "character locations" and not simply "characters. As we will find out further below and in *Chapters 14*, *20* and *Appendix A*, some characters are non-printable and as a matter-of-fact perform special functions. The following little program prints out the entire *usable* character set:

```
10 FOR a=32 TO 255: PRINT CHR$ a;: NEXT a
```

At the top you can see a space, 15 symbols and punctuation marks, the ten digits, seven more symbols, the capital letters, six more symbols, the lower case letters and five more symbols. These are all (except £ and ©) taken from a widely-used set of characters known as *ASCII* (standing for American Standard Codes for Information Interchange); *ASCII* also assigns numeric codes to these characters, and these are the codes that the ZX Spectrum Next uses.

## The graphics symbols

The rest of the characters are not part of *ASCII*, and are specific to the ZX Spectrum Next. First amongst them are a *space* and 15 patterns of black and white blobs. These are called the *graphics symbols* and can be used for drawing rudimentary pictures. You can enter these from the keyboard, using what is called *graphics mode*.

If you press GRAPHICS then the cursor will change to a flashing white/magenta. Now the keys for the digits 1 to 8 will give the graphics symbols: on their own they give the symbols drawn on the keys; and with either shift pressed they give the same symbol but inverted, i.e. black becomes white, and vice versa. Regardless of shifts, digit 9 takes you back to normal mode (blue cursor) and digit 0 is DELETE. Here are the sixteen graphics symbols:

| Symbol | Code | Key | Symbol | Code | Key |
|---|---|---|---|---|---|
|  | 128 | 8 |  | 143 | Shift+8 |
|  | 129 | 1 |  | 142 | Shift+1 |
|  | 130 | 2 |  | 141 | Shift+2 |
|  | 131 | 3 |  | 140 | Shift+3 |
|  | 132 | 4 |  | 139 | Shift+4 |
|  | 133 | 5 |  | 138 | Shift+5 |
|  | 134 | 6 |  | 137 | Shift+6 |
|  | 135 | 7 |  | 136 | Shift+7 |

*Table 5 – Graphics Symbols*

## Tokens

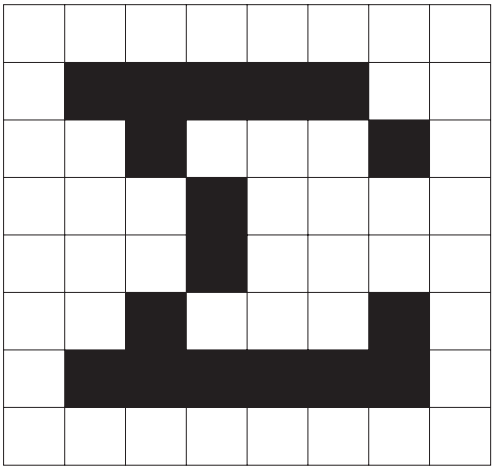
When not used as single symbols, several character codes are used in an alternative manner, where they are called *tokens*, Tokens represent whole words, such as PRINT, STOP, >=, <>, <= and so on. This is to save space in the machine's main RAM, thus maximising the available program space by substituting multiple character words for single character codes.

## BIN and USR

After the graphics symbols, you will see what appears to be another copy of the alphabet from A to U. These are characters that you can redefine yourself, although when the machine is first switched on they are set as letters – they are called *user-defined* graphics (or UDGs for short). You can type these in from the keyboard by going into graphics mode, and then using the letters keys from A to U.

To define a new character for yourself, follow this recipe – it defines a character to show the mathematical symbol Σ (Greek for Συνολο=*sum*).

i. Work out what the character looks like. Each character has an 8x8 square of dots, each of which can show either the paper colour or the ink colour (see *Chapter 15* regarding INK and PAPER). You'd draw a diagram something like this, with black squares for the ink colour:



We've left a 1 square margin round the edge because the other letters all have one (except for lower case letters with tails, where the tail goes right down to the bottom of the square).

ii. Work out which user-defined graphic is to show - let's say the one corresponding to S, so that if you press S in graphics mode you get Σ on your screen.

iii. Store the new pattern. Each *user-defined graphic* has its pattern stored as eight numbers, one for each row. You can write each of these numbers as BIN followed by eight 0s or 1s – 0 for paper, 1 for ink – so that the eight numbers for our character are:

```
BIN 00000000
BIN 01111100
BIN 00100010
BIN 00010000
```

```
BIN 00010000
BIN 00100010
BIN 01111110
BIN 00000000
```

(If you know about binary numbers, then it should help you to know that BIN is used to write a number in binary instead of the usual decimal.)

These eight numbers are stored in memory, in eight places, each of which has an address. The address of the first byte, or group of eight digits, is USR "S" (S because that is what we chose in (ii)), that of the second is USR "S"+1, and so on up to the eighth, which has address USR "S"+7.

USR here is a function to convert a string argument into the address of the first byte in memory for the corresponding *user-defined graphic*. The string argument must be a single character which can be either the user-defined graphic itself or the corresponding letter (in upper or lower case). There is another use for USR, when its argument is a number, which will be dealt with in subsequent chapters.

Even if you don't understand this, the following program will do it for you:

```
 5 FOR n=0 TO 7
10 READ row: POKE USR
   "S"+n,row
15 NEXT n
20 DATA BIN 00000000
25 DATA BIN 01111100
30 DATA BIN 00100010
35 DATA BIN 00010000
40 DATA BIN 00010000
45 DATA BIN 00100010
50 DATA BIN 01111110
60 DATA BIN 00000000
```

The above example can also be rewritten using integer variables without the use of BIN while still expressing the graphic matrix in binary form. Can you restate it per what you've learned?

## POKE and PEEK

The POKE statement stores a number directly in a memory location, bypassing the assignment (LET) mechanism normally used by *NextBASIC* which also tracks its place in memory. The opposite of POKE is PEEK, and this allows us to look at the contents of a memory location although it does not actually alter the contents of that location. They will be dealt with properly in *Chapter 23*.There are a few more efficient ways to type all the above but for now, we're using the simplest forms of PEEK and POKE.

The tokens (which we referred to a little earlier) are stored right after character code 128. As you saw, in the character set printing example, codes 0 to 31 were absent. These are *control* characters or as commonly referred to: *control codes*. They either don't produce characters on screen – although they do have an effect on what's printed there – or, alternatively, they are used to control something other than the display itself, and the screen displays ? to show that it doesn't understand them. They are described more fully in *Appendix A*.

Three *control codes* that are used with screen output, are those with codes 6, 8 and 13:

CHR$ 6 prints spaces in exactly the same way as a *comma* does in a PRINT statement; for instance:

```
PRINT 1; CHR$ 6;2
```

does the same as:

```
PRINT 1,2
```

Obviously this is not a very clear way of using it. A more subtle way is to say:

```
a$="1"+CHR$ 6+"2"
PRINT a$
```

CHR$ 8 is *backspace*: it moves the print position back *one* place – try:

```
PRINT "1234";CHR$ 8;"5"
```

which prints up:

```
1235
```

as 5 takes the place of 4 from the string printed in the first part of the PRINT statement. CHR$ 13 is *carriage return*: it moves the print position on to the beginning of the next line.

Effectively:

```
PRINT "1234";CHR$ 13;"5678"
```

is the same as:

```
PRINT "1234":PRINT "5678"
```

It may not be immediately apparent why you wouldn't do the latter but it's possible also to do:

```
a$="1234"+CHR$ 13+ "5678"
PRINT a$
```

In which case you can see the usefulness of a single *carriage return* character.

The screen also uses *control codes* 16 to 23; these are explained in *Chapters 13 and 14. All the control codes* are listed in *Appendix A*.

Using the codes for the characters we can extend the concept of *alphabetical ordering* to cover strings containing any characters, not just letters. If instead of thinking in terms of the usual alphabet of 26 letters we use the extended alphabet of 256 characters, in the same order as their codes, then the principle is exactly the same. For instance, these strings are in their ZX Spectrum Next alphabetical order: (Notice the rather odd feature that lower case letters come after all the capitals: so a comes after Z; also, spaces *matter*.)

```
CHR$ 3+"ZOOLOGICAL GARDENS"
CHR$ 8+"AARDVARK HUNTING"
"   AAAARGH!"
"(Parenthetical remark)"
"100"
"129.95 inc. VAT"
"AASVOGEL"
"Aardvark"
"PRINT"
"Zoo"
```

"[interpolation]"
"aardvark"
"aasvogel"
"zoo"
"zoology"

Here is the rule for finding out which order two strings come in. First, compare the first characters. If they are different, then one of them has its code less than the other, and the string it came from is the earlier (lesser) of the two strings. If they are the same, then go on to compare the next characters. If in this process one of the strings runs out before the other, then that string is the earlier, otherwise they must be equal.

The relations =, <, >, <=, >= and <> are used for strings as well as for numbers: < means *comes before* and > means *comes after*, so that:

"AA man"<"AARDVARK"
"AARDVARK">"AA man"

are both true.

<= and >= work the same way as they do for numbers, so that:

"The same string"<="The same string"

is *true*, but:

"The same string"<"The same string"

is *false*.

Experiment on all this using the program here, which inputs two strings and puts them in order.

```
10 INPUT "Type in two
   strings:", a$, b$
20 IF a$>b$ THEN a$,b$=b$,a$
30 PRINT a$;" ";
40 IF a$<b$ THEN PRINT "<";:
   GO TO 60
50 PRINT "=";
60 PRINT " ";b$
70 GO TO 10
```

Note how we are using a multiple assignment in order to swap a$ and b$ in line 20 as

```
a$=b$:b$=a$
```

would not have the desired effect since a$ would have the value of b$ prior to trying to assign its value to b$.

This program sets up user-defined graphics to show chess pieces:

P for *pawn*
R for *rook*
N for *knight*
B for *bishop*
K for *king*
Q for *queen*

## Chess pieces

```
  5 b,c,d=BIN 01111100,BIN
    00111000,BIN 00010000
 10 FOR n=1 TO 6: READ p$: REM
    6 pieces
 20 FOR f=0 TO 7: REM read
    piece into 8 bytes
 30 READ a: POKE USR p$+f,a
 40 NEXT f
 50 NEXT n
100 REM bishop
110 DATA "b",0,d, BIN
    00101000,BIN 01000100
120 DATA BIN 01101100,c,b,0
130 REM king
140 DATA "k",0,d,c,d
150 DATA c, BIN 01000100,c,0
160 REM rook
170 DATA "r",0, BIN
    01010100,b,c
180 DATA c,b,b,0
190 REM queen
200 DATA "q",0, BIN 01010100,
    BIN 00101000,d
210 DATA BIN 01101100,b,b,0
220 REM pawn
230 DATA "p",0,0,d,c
240 DATA c,d,b,0
250 REM knight
260 DATA "n",0,d,c, BIN
    01111000
270 DATA BIN 00011000,c,b,0
```

Note that 0 can be used instead of BIN 00000000.

When you have run the program, look at the pieces by going into graphics mode.

## Alternative Character Sets

As we are going to see in *Chapter 20 – Channels, Streams and Windows* the ZX Spectrum Next provides via its windowing system, the ability to display alternative character sets. In order to set up however an alternative character set, characters have to be defined somewhere in memory, very similarly to the way we did the chess pieces or the Σ symbol above. The characters redefined are limited to the 96 from code 32 until code 127 and should be in that order. A successive series of 768 POKE statements incrementing the memory address by one location at the time, will define them and then a last POKE altering the

CHARS system variable (See *Chapter 24 – System Variables*) will point *NextBASIC* to the location of this new character set.

## Character Graphics Mode

In the following chapter, we will be introduced to *Layer 3* – the Character Graphics mode; this is a hybrid graphics mode based around the notion of a character *tile,* that is to say an 8 × 8 pixel matrix very much like the ones we explored above with User Defined Graphics with four very crucial differences:

- Each character tile can have up to sixteen colours and not only two.
- *All* ASCII characters can be defined by tiles giving the user in effect a truly multi-lingual character display.
- *Layer 3* displays can be either 80 columns by 32 rows or 40 columns by 32 rows and not only 32 columns by 24 rows as the regular Spectrum display is.
- *Layer 3* cannot be accessed from *NextBASIC* (at the time of writing) in the same, straightforward way, other modes/layers are. You will need to write functions and procedures that utilise the PEEK, POKE, IN, OUT and REG facilities as well as the BANK commands at your disposal in order to make use of this powerful mode.

*Layer 3* has other uses as well and we will be discussing those in the following 3 chapters.

## Exercises

1. Imagine the space for one symbol divided up into four quarters like a Battenburg cake. Then if each quarter can be either black or white, there are 2 x 2 x 2 x 2=16 possibilities. Find them all in the character set.

2. Run this program:

```
10 INPUT a
20 PRINT CHR$ a;
30 GO TO 10
```

   If you experiment with it, you'll find that CHR$ a is rounded to the nearest whole number; and if a is not in the range 0 to 255 then the program stops with error report:

   B integer out of range.

3. Which of these two is the lesser?

   "EVIL"
   "evil"

# Chapter 14 – More about PRINT and INPUT

## Coordinate Systems

Before we go into more detail about how we can exercise a bit more control on **PRINT** and **INPUT** it is useful to understand a little bit about the way *NextBASIC* views character positioning on the screen. Due to the requirements for backwards compatibility with previous Sinclair computers, *NextBASIC* uses two distinct coordinate systems to keep track of where text is input or outputted. The first –or legacy– system is based on a virtual matrix that exists on screen and organises it in rigid rows and columns. The second, is more precise and allows for freely positioned columns and rows along the x and y-axes. Additionally the legacy coordinate system has been extended to allow for direct manipulation of the footer bar and status area for layers other than *Layer 0,* which is not normally possible in the legacy system.

## Screen Modes and Pixel Coordinates

In order to make a concrete distinction between the two coordinate systems, we should first discuss a little bit about the ZX Spectrum Next's display system. We will revisit this again in *Chapters 15* and *16* in more detail as these chapters deal with the full graphics capabilities of the computer rather than the subset dedicated to screen character manipulation, but for now let's enumerate the screen modes in a simple fashion.

The ZX Spectrum Next has 12 distinct graphics modes broken into 4 groups –or *layers*– with an additional Sprite Layer which we will not be covering in this chapter . These modes are accessed using the **LAYER** command with the exception of *Layer 3* (*Character Graphics)* and the higher resolution *Layer 2* modes and they are the following*:*

- Layer 0
  - *Layer 0* – Standard Spectrum (ULA) mode, 256 w x 192 h pixels, 8 colours total (2 intensities), 32 x 24 cells, each capable of displaying 2 colours
- Layer 1
  - *Layer 1, 0* – LoRes (EnhancedULA) mode, 128 w x 96 h pixels, 256 colours total, 1 colour per pixel
  - *Layer 1, 1* – Standard Res (EnhancedULA) mode, 256 w x 192 h pixels, 256 colours total, 32 x 24 cells, each capable of displaying 2 colours
  - *Layer 1, 2* – Timex HiRes (EnhancedULA) mode, 512 w x 192 h pixels, 256 colours total, only 2 colours on screen
  - *Layer 1, 3* – Timex HiColour (EnhancedULA) mode, 256 w x 192 h pixels, 256 colours total, 32 x 192 cells, each capable of displaying 2 colours
- Layer 2
  - *Layer 2* – 256 w x 192 h pixels, 256 colours total, one colour per pixel
  - *Layer 2,2 – 320 w x 256 h pixels, 256 colours total, one colour per pixel*
  - *Layer 2,3 – 640 w x 256 h pixels, 16 colours total, one colour per pixel*
- Layer 3
  - *Layer 3,0* – Text mode, 320 w x 256 h pixels, 256 colours total, 40 x 32 cells each capable of displaying 2 colours
  - *Layer 3,1* – Text mode, 640 w x 256 h pixels, 256 colours total, 80 x 32 cells, each capable of displaying 2 colours
  - *Layer 3,2* – Graphics mode, 320 w x 256 h pixels, 256 colours total, 40 x 32 cells each capable of displaying 16 colours
  - *Layer 3,3* – Graphics mode, 640 w x 256 h pixels, 256 colours total, 80 x 32 cells, each capable of displaying 16 colours

*Layers 2,2* as well as *2,3* together with *Layer 3* are not currently available to **PRINT** and **INPUT** and therefore won't be discussed in this chapter; they are mentioned here for completeness.

Technically speaking, *Layer 1,1* is the same as *Layer 0* with extra colour capabilities however *NextBASIC* treats them differently to maintain a consistent way of addressing the extra capabilities of the ZX Spectrum Next's *EnhancedULA*. The legacy coordinate system we discussed above applies only on *Layer 0*, whereas *Layers 1* and *2* use the new system.

There are three major differences between *Layer 0* and *Layers 1* and *2* as far as character positioning goes. There are more differences but we will examine these in turn in the special graphics *Chapters 15 – 17.* These are:

1. *Layer 0* is organised in a strict 32 columns by 24 rows matrix while the rest can both position characters on a similar matrix (according to character size), or, if so desired, anywhere along the *y* and *x* axes.
2. The user cannot –normally– position characters on the two bottom rows of the *Layer 0* screen while this is possible in the other *layers*.
3. *Layer 0* pixel coordinates begin at the bottom left corner and extend up and to the right while for the rest of the *layers*, pixel coordinates begin at the top left corner and extend down and to the right. This particular difference is not important for character placement on *Layer 0* but it is for the rest of the *layers* and definitely, as we are going to see further down this manual, extremely important for positioning graphics.

## Changing the size of characters

With the exception of *Layer 0*, which has, as we mentioned, a rigid organisation of character positions on screen in a 32 x 24 character matrix, all other *layers* have the ability to position characters either rigidly as above (ie. in a *rows* x *columns* matrix) or freely according to *pixel position* of each character matrix's top left corner.
Character size can be modified horizontally with the following sequence:

PRINT CHR$ 30; CHR$ *n*;

where *n* can be a number from *3* to *8*, which sets the width of all characters displayed on screen from a minimum of **3** to a maximum of **8** pixels wide. Character size is modified vertically by issuing:

PRINT CHR$ 29; CHR$ *n*;

where *n* can be a number from *0* to *3*, which sets the height of all characters displayed on screen to the following predetermined heights in pixels:

| Value of *n* | Size (pixels) | Description |
|---|---|---|
| 0 | 8 | Normal Size |
| 1 | 16 | Double Size |
| 2 | 6 | Reduced Size |
| 3 | 12 | Double Reduced Size |

These sequences which are more appropriately called *control codes,* are character size shortcuts for *text windows.* These can also be used on *Layer 0* but you would need to *open a window* first when in that mode. The rest of the *layers* have predefined and pre-opened *full-screen text windows* and therefore these *control codes* work there by default. We will discuss *text windows* at length in *Chapter 20 – Channels, Streams and Windows* so for now keep these two *control codes* in mind as only working outside *Layer 0.* They are extremely important to know, as they modify the behaviour of the **AT** and **TAB** modifiers we will examine below.

## Using AT to print to a certain location

You have already seen **PRINT** used quite a lot, so you will have a rough idea of how it is used. Expressions whose values are printed are called *PRINT items*, and they are separated by commas, semicolons and apostrophes, which are called *PRINT separators*. A *PRINT item* can also be nothing at all, which is a way of explaining what happens when you

use two commas in a row.
There are two more kinds of *PRINT items*, which are used to tell the computer not what, but where to print. For example **PRINT AT 11,16;"*"** prints a star in the middle of the screen in *Layer 0*.

The modifier

**AT** *vertical_position*, *horizontal_position*

moves the **PRINT** position (the place where the next item is to be printed) to the vertical and horizontal position specified. Horizontal positions are measured in *columns* and vertical positions in *rows* however for layers other than *Layer 0*, the number of columns and rows varies according to the size of characters used (and for *HiRes* mode the horizontal resolution as well). Character sizes are set according to the previous *section*, however for **AT** usage purposes, we need to note that *double-width and double-height* character sizes do not modify the maximum *columns* and *rows* **AT** will accept as parameters, so if for example you use **PRINT CHR$ 29; CHR$ 1** for characters that are **16** pixels high, you will still get a maximum of **24** rows for **AT** purposes.

You may have noticed at the beginning of this chapter that we discussed *Layer 0* as being organised for character printing purposes, in a matrix of 24 rows by 32 columns. As you will see however when in *Layer 0*, *NextBASIC* will not give you access to the last two rows since, as we discussed in *Chapter 1*, the bottom two rows of the screen are reserved. This is also true for bitmap graphics commands as you will see in *Chapters 17* and *18*. We will expand further on the possible combinations for **AT** but for now give the command:

```
PRINT AT 22,31;"*"
```

and you will immediately receive error **5 Out of screen, 0:1**. It's not difficult to understand why that happened. As we can see in *Fig. 8* below, for the purposes of printing via *NextBASIC*[1], your computer has a vertical resolution of 192 pixels. Since, as we learned in *Chapter 13*, each character is 8 pixels high, we can make a quick division and see that $192 \div 8 = 24$. Knowing that the two last lines are reserved and not accessible to us, we can reduce our available rows by a further 16 pixels (or 2 rows) so we get a total 22 rows. As your computer starts counting from *zero*, 22 rows would go up to 21 as a value, which in turn explains why you received the error.

Rows on which we can place output using **AT**, are numbered therefore from **0** (at the top) to **21**, and columns from **0** (on the left) to **31**.

This situation changes when we change *layers* and go to the other two groups (remember that *Layer 3* and the higher resolution *Layer 2* sublayers are excluded). As discussed previously, columns and rows on these are calculated according to the width of characters that we have selected with the *control codes*. Before we illustrate graphically how the screen is organised, the following table will give you the possible combinations in columns per character width. Remember that you can also figure this out on your own by dividing the maximum resolution of the *layer* you're using by the selected character width.

| | Number of columns per Layer | | |
|---|---|---|---|
| Character width (in px) | LoRes Layer 1,0 (128 x 96) | HiRes Layer 1,2 (512 x 192) | Standard Res Layers: 1,1–1,3 – 2 (256 x 192) |
| 3 | 42 | 170 | 85 |
| 4 | 32 | 128 | 64 |
| 5 | 25 | 102 | 51 |
| 6 | 21 | 85 | 42 |
| 7 | 18 | 73 | 36 |
| 8 | 16 | 64 | 32 |

*Table 6 – Column positions for PRINT according to character size*

1 *The maximum screen resolution of the ZX Spectrum Next is 320 x 256 pixels (or 640 x 256 half-width pixels), however these resolutions are only available to Layers 2,3 and Sprite Layers as we will see in the following chapters.*

*Table 6* above, showed us that although we could pack our screen with 170 characters per line, in practice 3 pixel wide fonts are almost unreadable, even at the highest available resolution of *Layer 1,2*. In the example program that's meant to demonstrate character cells for the AT modifier (but written using the POINT modifier strangely enough!) we're including below, you can see all the possible combinations for all *layers*.

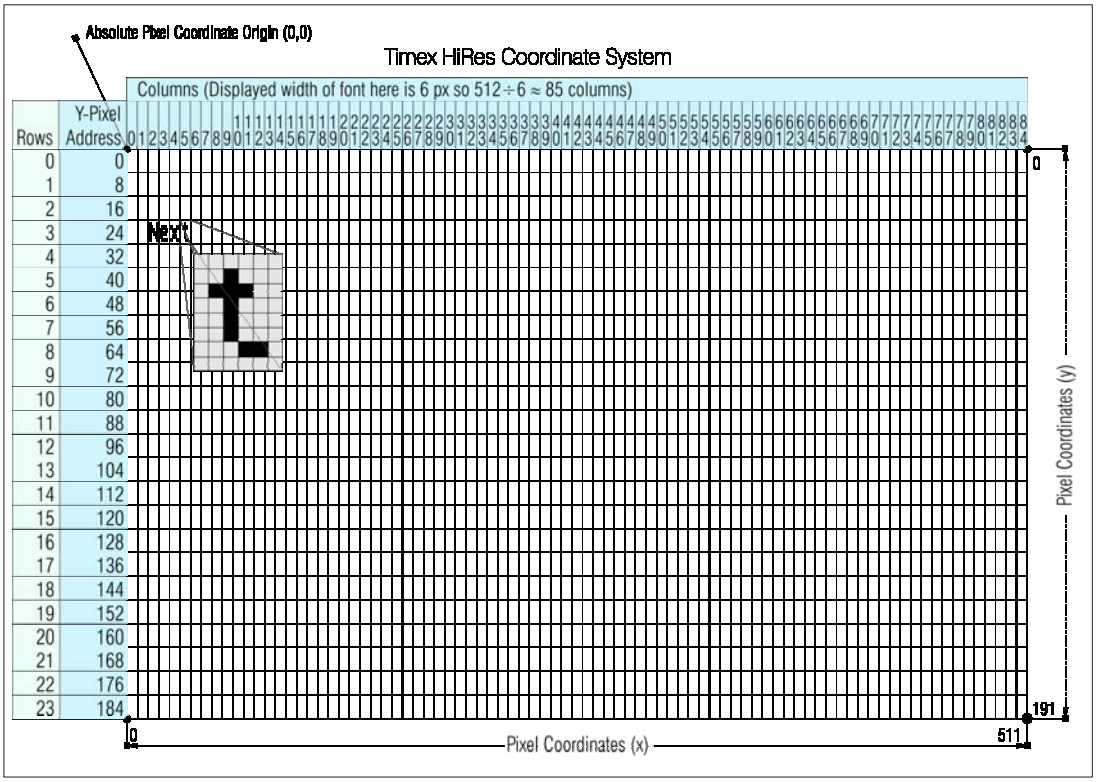



*Fig. 8 – Layer 0 coordinate system for PRINT and INPUT*

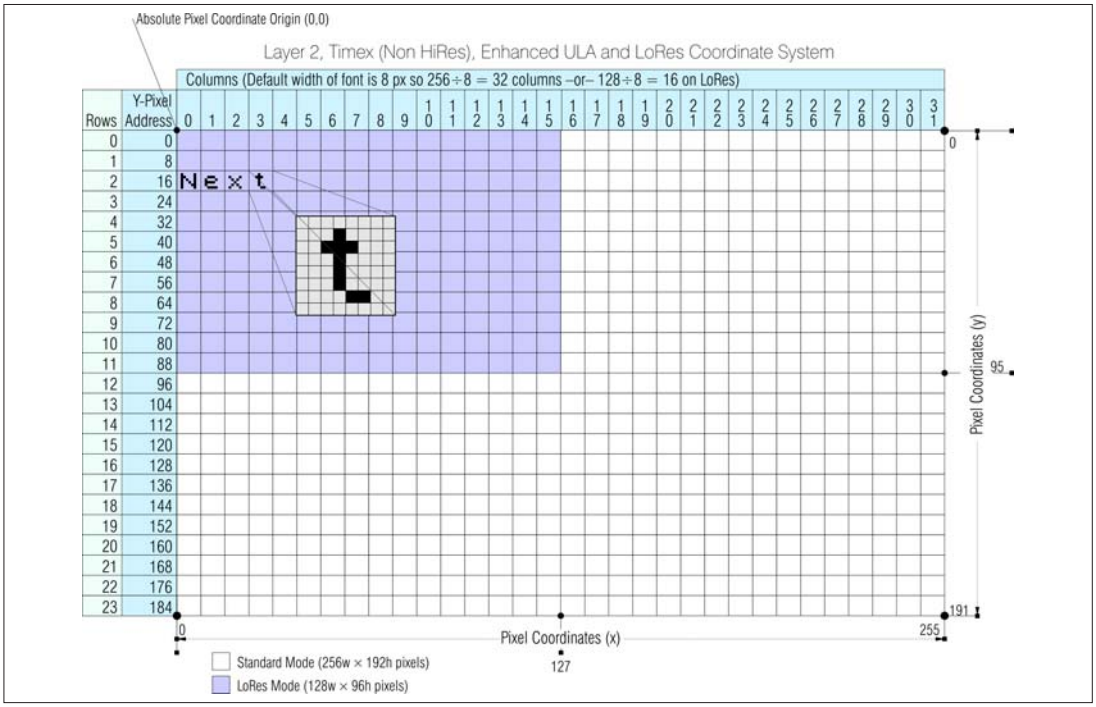



*Fig. 9 – LoRes and Standard Resolution coordinate system for PRINT and INPUT*

The author's personal preference is the 128 column text of *HiRes Layer 1,2* as it's clear enough to read but not too big as to not be able to fit a lot of information onto your screen.

## Using POINT to print to a certain location

In *Fig. 9* above, we see the main difference between *PRINT items* on *Layer 0* and the other *layers* and that's none other than the previously mentioned ability to place them in any X and Y coordinate we please. This diagram assumes a standard 8x8 character size but where you only saw rows in *Fig. 8*, here you also see a pixel value. This corresponds to the placement of each row and column in *Layer 0* but in fact, it could be anything within the boundaries of the horizontal and vertical resolution. Let's switch *layers* and try to do the same thing:

```
LAYER 1,1:PRINT POINT 248,176;"*"
```

Unlike before you'll will not get an **5 Out of screen, 0:1** error and you will get an asterisk at the rightmost edge of the screen like we expected to get the first time we gave the **PRINT AT 22,31** command. The two values correspond to **22** times the **character height** and **31** times the **character width** (both of which are **8** pixels). You can see at the same time the notion of the *free* placement of characters as the addressing of the location is now in pixels and not the fixed rows and columns. What's also immediately visible is that addressing the location on screen in pixel coordinates is different as it reverses the order of the location parameters from *y,x* to *x,y* and that's done to match the syntax of the rest of the graphics commands that accept pixel coordinates as parameters. To replicate the behaviour of the first **PRINT AT** command on *Layer 0* and get an error, we will need to place the output of print, outside the boundaries of the screen like so:

```
LAYER 1,1: PRINT POINT 256,0;"*"
```

would produce the same exact error. To properly calculate where to print if you want to keep your coordinates cell-based instead of pixel-based, a simple *function* could do that for you quite easily. In *Fig. 8* as well as *Fig. 9* we've done that for you assuming a standard font, but what about a shorter, or perhaps taller font? It's quite simple if you keep in mind that, if you follow the heights defined earlier, you can find exactly how many rows and columns you can fit in your screen. Note that **POINT**'s arguments must not begin with a parenthesis because it will be evaluated as a function and attempting to store the line you're typing will fail.

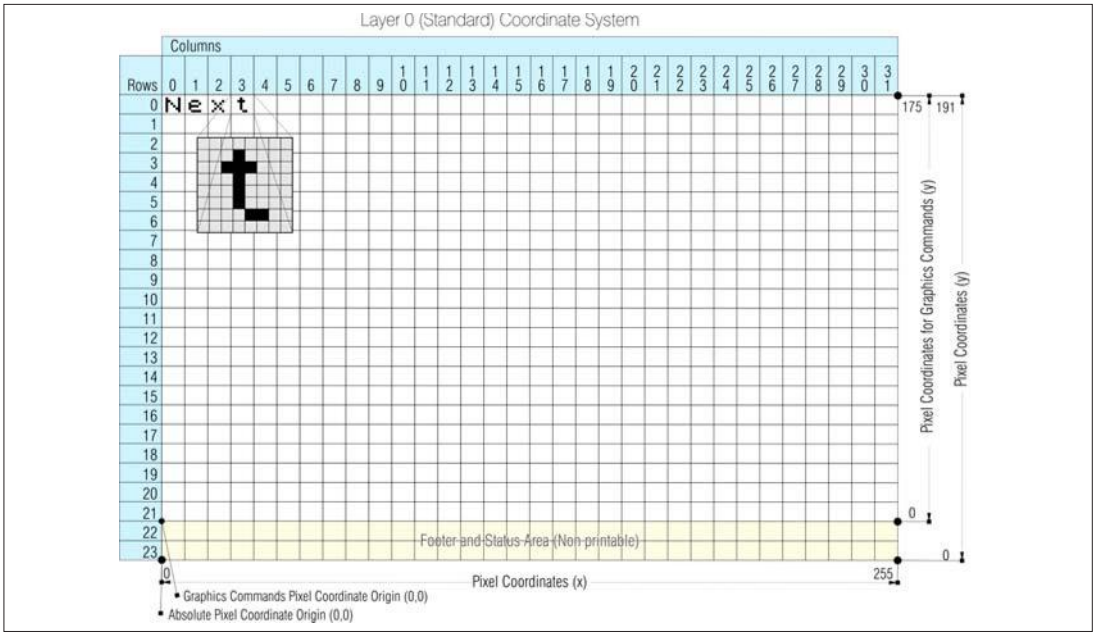



*Fig. 10 – High Resolution coordinate system for PRINT and INPUT*

The following –very slow– program demonstrates exactly how things are positioned on screen with every change in *Layer* and furthermore gives you some insight on how **PRINT POINT** as well as –indirectly– **PRINT AT** is affected every time your screen mode changes. Try to walk through the program to figure out how it operates:

```
 10 REM First we disable LAYER
    2 and then we set Standard
    ULA Display Mode
 20 LAYER 2,0
 30 LAYER 0
 40 MaxX,MaxY=128,96
 70 mul,div,add=1,1,0
100 chsz,h=8
120 FOR m=0 TO 5
130 n,d=0,1
150 IF m=0 THEN GO TO 370: REM
    Layer 0  not supported by
    PRINT POINT
160 FOR a=3 TO 8
180 FOR b=0 TO 3
190 n,d=0,1
210 PROC LayChange(m,a,b)
220 FOR r=0 TO (MaxY*mul)-1
    STEP h
230 FOR c=0 TO (MaxX*mul)-chsz
    STEP chsz
235 row = (r+add)/div
240 IF r=0 AND c<>0 THEN PRINT
    POINT
    c,row;d : d+=1
250 IF c=0 AND r=0 THEN PRINT
    POINT
    c,row;n : n+=1
260 IF c=0 AND r<>0 THEN PRINT
    POINT
    c,row;n :n+=1
270 IF c<>0 AND r<>0 THEN
    PRINT POINT
    c,row;"*"
280 IF n=10 THEN n=0
290 IF d=10 THEN d=0
300 NEXT c
310 IF c=1 THEN n=0
320 NEXT r
330 PAUSE 0
340 IF m=0 THEN GO TO 370
350 NEXT b
360 NEXT a
```

```
 370 NEXT m
 380 LAYER 0
 390 LAYER 2,0
 400 STOP
1000 DEFPROC LayChange(mode,ch,he)
1010 div,add, maxX, maxY, mul,
     chsz=1,0,128,96,2,ch
1070 IF he=0 THEN h=8
1080 IF he=1 THEN h=16
1090 IF he=2 THEN h=6
1100 IF he=3 THEN h=12
1110 REM Layer 0 is not covered
     as PRINT POINT doesn't work
1120 IF mode=1 THEN LAYER 1,0:
     CLS:mul=1:PRINT CHR$ 30;
     CHR$ ch:PRINT CHR$ 29; CHR$
     he:PRINT AT 0,0;"LoRes"'"
     CSIZE (HxW)  ";h;" x
     ";chsz'"PRESS ANY KEY":
     PAUSE 0:CLS:ENDPROC
1130 IF mode=2 THEN LAYER 1,1:
     CLS : PRINT CHR$ 30; CHR$
     ch:PRINT CHR$ 29;CHR$ he:
     PRINT AT
     0,0;"EnhancedULA"'"CSIZE
     (HxW)  ";h;
     " x ";chsz'"PRESS ANY KEY"
     :PAUSE 0:CLS:ENDPROC
1140 IF mode=3 THEN LAYER 1,2:
     CLS:MaxX=256:PRINT CHR$ 30;
     CHR$ ch:PRINT CHR$ 29; CHR$
     he:PRINT AT 0,0;"Timex
     HiRes"'"CSIZE (HxW)  ";h;"
     x ";chsz'"PRESS ANY KEY":
     PAUSE 0:CLS:ENDPROC
1150 IF mode=4 THEN LAYER 1,3:
     CLS:PRINT CHR$ 30; CHR$ ch:
     PRINT CHR$ 29; CHR$ he:
     PRINT AT 0,0;"Timex
     HiColour"'"CSIZE (HxW)
     ";h;" x ";chsz'"PRESS ANY
     KEY":PAUSE 0:CLS:ENDPROC
```

```
1160 IF mode=5 THEN LAYER 2,1:
     CLS:PRINT CHR$ 30;CHR$
     ch:PRINT CHR$ 29; CHR$
     he:PRINT AT
     0,0;"Layer2"'"CSIZE (HxW)
     ";h;" x ";chsz'"PRESS ANY
     KEY":PAUSE 0:CLS:ENDPROC
```

## SCREEN$

SCREEN$ is the reverse function to PRINT AT, and will tell you (within limits) what character is at a particular position on the screen. It uses line and column numbers in the same way as the *Layer 0* version of PRINT AT, but enclosed in parentheses. For instance:

```
PRINT SCREEN$ (11,16)
```

will retrieve the star you printed in the first example of the previous section. SCREEN$ *only works on Layer 0* and will return everything printed there, even if you switch *layers* during the process as long as the memory used (which is *shared* between *Layers 0, 1* and *3* as you will see in *Chapter 23)* has not been overwritten by another display related command. *Type:*

```
10 LAYER 0:PRINT AT 11,11;"*"
20 LAYER 1,0:PRINT AT
   0,0;SCREEN$ (11,11)
```

You will get a huge * on the upper left corner of your screen even if the original * is not visible anymore on screen. Changing line 10 to LAYER 1,0 from LAYER 0 will produce a *null* string.

Characters taken from tokens print normally, as single characters, and spaces return as spaces. Lines drawn by PLOT, DRAW or CIRCLE, user-defined characters and graphics characters return as a *null* (empty) string, however. The same applies if OVER (See *Chapter 16*) has been used to create a composite character. The way that SCREEN$ works is that it matches the character in a screen location to the bitmapped image of the character in the ROM of *NextZXOS*. If they match it will return it. If the picture in the location doesn't match any known character it will return an empty string.

## TAB

If you're familiar with word processing, other computers, or even typewriters, you may be also familiar with the concept of a *tab*, or *tabulating* character. What this does in other computers is to insert a special character which will move the cursor right by a predetermined amount of locations in order to arrive to a specific column in your text. The ZX Spectrum Next, doesn't quite work like this although the ending result on your screen is pretty much equivalent. The modifier:

TAB *column*

prints enough spaces to move the PRINT position to the column specified. It stays on the same line, or, if this would involve backspacing, moves on to the next one. Note that the computer reduces the column number *modulo X* with *X* being the maximum amount of columns available per the width of character chosen for each *Layer* (meaning it divides by X and takes the remainder); so for example for *Layer 0*, TAB 33 means the same as TAB 1.

The code:

```
PRINT TAB 30;1;TAB 12;"Contents"; AT
3,1;"CHAPTER";TAB 24;"page"
```

demonstrates, how you might print out the heading of a contents page on page 1 of a book (if that book was displayed using ZX Spectrum Next characters of course!)

Try running this:

```
10 FOR n=0 TO 20
20 PRINT TAB 8*n;n;
30 NEXT n
```

This shows what is meant by the TAB numbers being reduced *modulo X*. For a more elegant example, change the 8 in line 20 to a 6 or even try to implement this on a different *layer* such as the *HiRes* one as it allows more room for demonstration of this functionality by adding LAYER 1,2 before line 10.

As you'll see in *Chapter 20,* TAB accepts a two-byte parameter which means it accepts a maximum column number of 65535! Not that you'd ever want to use that!

Some small points:

1. These new items are best terminated with semicolons, as we have done above. You can use commas (or nothing, at the end of the statement), but this means that after having carefully set up the PRINT position, you immediately move it on again which wouldn't usually be terribly useful.
2. As a reminder, you cannot print on the bottom two rows (22 and 23) on the *Layer 0* screen because they are reserved for commands, INPUT data (see below), reports/errors and so on. References to the *bottom line* usually mean line 21 and only apply to *Layer 0*.
3. You can use AT to put the PRINT position even where there is already something printed; the old stuff will be obliterated when you print more.

## CLS

Another statement that's connected with PRINT (although it's not *only* limited to it), is CLS. This clears the *whole screen*, something that is also done by CLEAR and RUN. The LAYER command does not clear the screen however, although it may switch to a new screen that has nothing on it. *Do not* assume a *Layer* is free of stuff just because you haven't used a command that outputs something on screen. Always give CLS after switching *layers* if you want to ensure a screen free of anything on it.

## Scrolling

When the printing reaches the bottom of the screen, the latter moves its contents upwards, to clear room on the bottom for new content. You can see this if you go into the status area by using the *Edit menu* option *Screen* and then type:

```
CLS:FOR n=1 TO 22:PRINT n:NEXT n
```

and then do:

```
PRINT 99
```

a few times.

Depending on the *layer* you are on, the computer may pause its screen output for you to review the content being printed and ask you a question or may simply display a block cursor at the lower right corner and wait.

On *Layer 0*, if the computer is printing out reams and reams of stuff on screen, it asks you before continuing. You can see this happening if you type:

```
CLS:FOR n=1 TO 100:PRINT n:NEXT n
```

When it has printed a screenful, it will stop, writing scroll? at the bottom of the screen. You can now inspect the first 22 numbers at your leisure. When you have finished with them, press y (for *yes*) and the computer will give you another screen full of numbers. Actually, any key will make the computer carry on except n (for *no*), SYMBOL SHIFT and A (for STOP as you can see printed on your ZX Spectrum Next's keyboard[2]), SPACE, BREAK (or CAPS SHIFT and SPACE) or Esc (the latter if you have a PS/2 type keyboard) . These will make the computer stop running the program with a report D BREAK - CONT repeats. On other *layers*, the scroll? message is replaced by a block cursor (called the *scroll prompt cursor*) at the lower right corner. The only keys which will stop the scrolling in *layers* other than 0 are the Esc key if on a PS/2 keyboard or the BREAK key (CAPS SHIFT and SPACE). Everything else will scroll the screen.

## Expanding on INPUT

The INPUT statement can do much more than we have told you so far. You have already seen INPUT statements like:

```
INPUT "How old are you?", age
```

in which the computer prints the caption How old are you? at the bottom of the screen, and then you have to type in your age.

In fact, an INPUT statement is made up of items and separators in exactly the same way as a PRINT statement is, so How old are you? and age are both *INPUT items*. *INPUT items* are generally the same as *PRINT items*, but there are some very important differences:

First, an obvious extra *INPUT item* is the variable whose value you are to type in – age in our example above. The rule is that if an *INPUT item* begins with a letter, it must be a variable whose value is to be input.

Second, this would seem to mean that you can't print out the values of variables as part of a caption; however, you can get round this by putting parentheses around the variable. Any expression that starts with a letter must be enclosed in parentheses if it is to be printed as part of a caption.

Any kind of *PRINT item* that is not affected by these rules is also an *INPUT item*. Here is an example to illustrate what's going on:

```
myage=INT(RND(100)):INPUT("I am ";myage;
". ");"How old are you?", yourage
```

myage is contained in parentheses, so its value gets printed out. yourage is not contained in parentheses, so you have to type its value in.

If you are in *Layer 0*, everything that an INPUT statement writes goes to the bottom part of the screen, which acts somewhat independently of the top half. In particular, its rows are numbered relative to the top line of the bottom half, even if this has scrolled the actual screen up (which it does if you type lots and lots of INPUT data).

To see how AT works in INPUT statements, try running this on *Layer 0*:

```
10 INPUT "This is line
   1.",a$; AT 0,0;"This is
   line 0.",a$; AT 2,0;
   "This is line 2.",a$; AT
   1,0; "This is still line
   1.",a$
```

2 *This functionality comes from the original ZX Spectrum single key (or tokenised) entry and it's retained for compatibility reasons.*

(Just press ENTER each time it stops.) When This is line 2. is printed, the lower part of the screen moves up to make room for it; but the numbering moves up as well, so that the rows of text keep their same numbers.

Now try this (again on *Layer 0*):

```
10 FOR n=0 TO 19: PRINT AT
   n,0;n;: NEXT n
20 INPUT AT 0,0;a$; AT 1,0;a$;
   AT 2,0;a$; AT 3,0;a$; AT
   4,0;a$; AT 5,0;a$;
```

As the lower part of the screen scrolls up and up, the upper part is undisturbed until the lower part threatens to write on the same line as the PRINT position. Then the upper part starts scrolling up to avoid this.

The other *layers* work in the same manner as described for *PRINT items*, that is in both rigid (cell matrix) and flexible (pixel coordinate) terms. To illustrate the difference, issue a LAYER 1,1 direct command and then modify the first example by first copying line 10 to line 20 and then changing all AT statements to POINT statements switching the x and y positions around, thus making the latter two parameters 0,16 and 0,8 respectively to reflect the height of characters (remember that on *layers* other than 0 character matrices will change according to character size and pixel positioning according to max resolution). The first thing you'll notice is that INPUT takes place at the top left of the screen as would with PRINT and the second one that the first *INPUT item* is NOT printed at "line" 1 but rather at "line" 0. Finally you can see from the modified first example that INPUT accepts a POINT modifier for positioning exactly like PRINT does.

## LINE input

Another refinement to the INPUT statement that we haven't seen yet is called LINE input and is a different way of inputting string variables. If you write LINE before the name of a string variable to be input, as in:

```
INPUT LINE a$
```

then the computer will *not* give you the string quotes that it normally does for a string variable, although it will pretend to itself that they are there. So if you type in:

```
Simon
```

as the INPUT data, a$ will be given the value Simon. Because the string quotes do not appear on the string, you cannot delete them and type in a different sort of string expression for the INPUT data. Remember that you cannot use LINE for numeric variables.

## Using Expressions for INPUT

There's an interesting capability of INPUT. While typing into an INPUT request that's expecting a number variable, you can use numeric expressions which can include previously defined variables. Try running this program:

```
10 a=14
20 INPUT numbers
30 PRINT numbers
40 GO TO 20
```

Input a few numbers, and they'll be printed as expected on the screen. Now type a and if you press ENTER, then 14 will appear! Try typing a+2 and 16 will appear. However, if you

type a variable name not previously defined then the computer will stop with the report 2 Variable not found, 20:1.

## Using control codes with PRINT

In the beginning of this chapter, we saw the effect that *control codes* 29 and 30 had in adjusting the size of the font that's currently printed on screen. There are more *control codes* that we can use with PRINT. CHR$ 22 and CHR$ 23 affect printing in the same manner as AT and TAB. They are rather odd as *control codes*, because whenever one is sent to the screen to be printed, it must be followed by two more characters that do not have their usual effect: they are treated as numbers (their codes) to specify the y and x positions (for AT) or the tab position (for TAB). You will almost always find it easier to use AT and TAB in the usual way rather than the control codes, but they might be useful in some circumstances. The AT control character is CHR$ 22. The first character after it specifies the y-position (be it a line number or y-pixel value according to the *layer* we're currently in) and the second the column number, so that:

```
PRINT CHR$ 22+CHR$ 1 +CHR$ c;
```

has exactly the same effect as:

```
PRINT AT 1,c;
```

This is so even if CHR$ 1 or CHR$ c would normally have a different meaning (for instance if c=13); the CHR$ 22 before them overrides that.

The TAB control character is CHR$ 23 and the two characters after it are used to give a number between 0 and 65535 specifying the number you would have in a TAB modifier:

```
PRINT CHR$ 23+CHR$ a+CHR$ b;
```

has the same effect as:

```
PRINT TAB a+256*b;
```

As with the character size *control codes*, there are further *control codes* that only apply to *layers* other than 0 and further modify their behaviour. One of those, is CHR$ 26 or the *Scroll-prompt inhibitor control code*. Set by CHR$ 26; CHR$ *n*; where *n* is the number of lines that can be scrolled off before the *scroll prompt cursor* appears (as discussed in the *Scrolling* section above) but after the first full screen length has been printed. If *n=0*, the scroll prompt function is inhibited for that *layer/window*. Note that the *n* number of lines is calculated based on an 8 pixel character height. That can lead to some very confusing results if your chosen character height is different. Some are easy to calculate like the standard or double height characters, with the latter in essence halving the amount of lines but others not so easy as with the reduced height and double reduced height characters. In the two last cases you have to calculate how many pixels your program outputs vertically by getting the amount of actual lines *times* the height of the characters and then divide the product by 8 (standard character height) in order to arrive to how many lines you need to instruct the system via the *Scroll-prompt inhibitor control code* to allow.

If this sounds unnecessarily complicated that's because it is! In most cases, the average user will either need to disable scroll-prompting by setting *n* to 0 or just set it to a full screen of data by setting *n* to 24 (for all screen modes except LAYER 1,0 which requires *n* set to 12).

On *Layer 0* you can duplicate that behaviour albeit in a less confusing way since the characters are always 8 pixels high, by employing a bit of POKE trickery to inhibit the scroll? prompt by doing:

```
POKE 23692,x
```

where **x** is the amount of lines the scroll prompt should be inhibited for –or in other words, *every time the scroll counter has been reached*. After this it will scroll up **x** number of times before stopping again with **scroll?**. As an example, try:

```
10 POKE 23692, 255
20 FOR n=1 TO 400
30 PRINT "line ";n
40 NEXT n
```

and watch everything whizz off the screen up until line 277 before the prompt to scroll re-appears! The technical explanation of what this **POKE** does, is that it modifies the *System Variable* **SCR CT**. It's important to also note that the Editor resets this *System Variable* so entering the **POKE** directly will have no appreciable effect on scrolling on *Layer 0* until it's entered in a program. We will examine all the possible combinations of **PRINT** *control codes* on *Chapter 21.* You will find more information about *System Variables* in *Chapter 24* and for **POKE** in *Chapter 23 – The Memory.*

## INKEY$

There's an additional function related to keyboard entry called **INKEY$**. **INKEY$** (which takes no argument) reads the keyboard immediately when it's invoked. If you are pressing exactly one key (or a **SHIFT** key and just *one* other key) then the result is the character that that key gives in that typing mode; otherwise the result is the empty string.
Try this program, which works like a typewriter.

```
10 IF INKEY$ <>"" THEN GO TO 10
20 IF INKEY$ = "" THEN GO TO 20
30 PRINT INKEY$;
40 GO TO 10
```

Here line **10** waits for you to lift your finger off the keyboard and line **20** waits for you to press a new key.

Unlike the regular **INPUT** (see also the next section), **INKEY$** doesn't wait for you. So you don't type **ENTER**, but on the other hand if you don't type anything at all then you've missed your chance. This also explains why the **GO TO** statements are needed in lines **10** and **20**.

## Using INPUT for game controllers

Much like **INKEY$** above, **INPUT** can also be used as a function with a numeric parameter *n* in order to read the current state of an input controller.

**INPUT** *n*

reads the current state of an input controller which can be one of the two joysticks (If *n* is 1 or **2**) or the *keyboard joystick*[3] (if *n* is **0**).

In each case, the value returned is a bitmask of the following value:

| | |
|---|---|
| bit **0** (value **1**) | *right* pressed |
| bit **1** (value **2**) | *left* pressed |
| bit **2** (value **4**) | *down* pressed |
| bit **3** (value **8**) | *up* pressed |
| bit **4** (value **16**) | *fire* pressed |

3 *NextZXOS has a feature where the keyboard can emulate one of the joystick standards it normally supports*

| | |
|---|---|
| bit 5 (value 32) | *fire2* pressed |
| bit 6 (value 64) | *fire3* pressed |
| bit 7 (value 128) | *fire4* pressed |

For example:

| | |
|---|---|
| `INPUT 1 & 8` | returns *false* (0) if up is not pressed on joystick 1, *true* (8 is non-zero) if it is |
| `INPUT 0 & @11110000` | returns *false* (0) if no fire buttons are pressed on joystick 0, *true* (non-zero) if at least one is |

The default *keyboard joystick* is set up to use the following keys:

| | |
|---|---|
| up | Q |
| down | A |
| left | O |
| right | P |
| fire | SPACE |
| fire2 | M |
| fire3 | ENTER |
| fire4 | X |

The *keyboard joystick* may be redefined with the INPUT function by if negative values are specified for *n*, as follows:

| | |
|---|---|
| `INPUT -1` | waits for a key to be pressed and assigns to *right* |
| `INPUT -2` | waits for a key to be pressed and assigns to *left* |
| `INPUT -3` | waits for a key to be pressed and assigns to *down* |
| `INPUT -4` | waits for a key to be pressed and assigns to *up* |
| `INPUT -5` | waits for a key to be pressed and assigns to *fire* |
| `INPUT -6` | waits for a key to be pressed and assigns to *fire2* |
| `INPUT -7` | waits for a key to be pressed and assigns to *fire3* |
| `INPUT -8` | waits for a key to be pressed and assigns to *fire4* |
| `INPUT -9` | (or any other negative value) clears all assignments |

The return value is the character code of the key pressed, which can be useful if you want to display the key just defined (although some special keys have codes below ASCII 32 which aren't PRINTable, so care should be taken). Here's an example on how to set up the keyboard joystick. Note that we cannot use INPUT to print on the screen so separate PRINT statements are needed!

```
100 x=INPUT -9:
110 ;clear the "keyboard joystick"
120 PRINT "Press a key for right"
130 x=INPUT -1
140 PRINT "Press a key for left"
150 x=INPUT -2
160 PRINT "Press a key for down"
170 x=INPUT -3
180 PRINT "Press a key for up"
190 x=INPUT -4
200 PRINT "Press a key for fire"
210 x=INPUT -5
220 REM Can leave additional fire
    buttons undefined if they aren't
    needed
```

# Chapter 15 – Colours

## An introduction to colour on the ZX Spectrum Next

Up until this point, we haven't really touched the subject of graphics manipulation on the ZX Spectrum Next and that's because the subject –mainly due to its original models' history– can be rather daunting to a beginner. As we've seen in *Chapters 1* and *14* where we really started to get into the more intricate details of the graphics system, the ZX Spectrum Next has some very interesting graphics capabilities that set it apart from its predecessors. The first capability which we will examine in depth is colour.

## Basics of computer colour

The first thing we need to remember, and that is important as it explains many of the design choices of the ZX Spectrum Next, is that at its heart beats an 8-bit[1] processor. This means that it is at its best when manipulating integer numbers up to 255 which are represented as 2 *to the power of* 8 – or properly written: $2^8$. Now taking a step back from that information we should concentrate on how colour can be represented. In reality there are many methods but the most common for a computer – and the one used by the ZX Spectrum Next – is to break colour into three components: Red, Green and Blue (or RGB) and to represent intensities of each of these components as numbers from 0 (for no intensity, or dark) to whatever maximum value a computer can store easily. In the ZX Spectrum Next's case each colour component can have 8 intensities making a total of 512 combined intensities which translates to 512 colours in total.

Now from basic maths, we know that to represent the number 8 in binary form (which is what computers understand) we can rewrite it as $2^3$ – or a binary number of 3 bits of length. To represent the total combination of colours when we combine the colour components, we can rewrite 512 as $(2^3)^3$ which in turn can be rewritten as $2^9$. This, given what we just said about the 8-bit nature of the ZX Spectrum Next is presenting a problem as the number of colours we have is represented by a 9-bit number while the computer can best manipulate efficiently 8 bits at a time. Keep this in mind for the moment and lets discuss how a colour could be represented in binary form.

## Colour organisation and representation

RGB colour has many ways of being stored in memory and it's usually denoted by the order of the bits. For example the BGR way stores first the bits for the Blue component, then the bits for the Green component and finally the bits for the Red component. As a matter of course, we usually add a number after every component (designated by a letter) to denote the number of bits (ergo also the number of intensities) or a single number at the end of the organisational acronym to denote that all components have equal number of intensities. For example R2G3B3 would mean an 8-bit colour organised as RGB with 2 bits (4 levels of intensity) on the Red component and 3 bits (8 levels of intensity) on the Green and Blue components.

The ZX Spectrum Next uses the GRB (for compatibility modes) and RGB methods of organisation and can store colour in three ways: G1R1B1, G3R3B2, R3G3B3 (or RGB3) and R3G3B2. The latter is really a shortcut for an 8-bit subset of the RGB3 way as we will see later but for now, let's assume it can manipulate 3-bit, 8-bit *and* 9-bit colours.

## Spatial vs Colour Resolution

Thus far, you've seen references about *resolution* when it comes to graphics but what does the word really mean? In short it means how much graphical information we can fit in a finite space. This doesn't actually mean how many dots we can fit in our screen (to make

1 *Bit is an acronym for BInary digiT and is a term used to describe the tiniest amount of information that a computer can hold, which is a single binary digit. Microprocessors are classified according to their ability to manipulate binary numbers of a certain order in one go. For example the Z80N CPU which is inside the ZX Spectrum Next can manipulate a number consisting of an 8 bit order in one go, so it is called an 8-bit microprocessor. By contrast the CPU inside the ZX Spectrum Next's "big brother", the Sinclair QL is a 32bit microprocessor as it can manipulate numbers consisting of 32-bits in one go.*

a gross simplification) but both *how many dots* and *how many colours* we can fit. The former is *spatial resolution* (it has one more component; *density* but this is not pertinent to this discussion) and the latter, *colour resolution*. It's important to make the distinction as we will see below because this informs not only a computer's design choices when it comes to graphics but also the special trickery that may be involved to display both on screen.

It's easy to understand *spatial resolution*. We –as you already read here and probably elsewhere– measure *spatial resolution* in *pixels* –or PICTure ELements–, in essence dots arranged in a Cartesian, two-dimensional coordinate system. Leaving colour information aside for the moment we can assign one bit per pixel and we can project this in the computer's memory in a linear fashion: Each horizontal line, follows the other so in the end we have a series of bits with each line being *w x n* times away from the very first bit that started our picture where *w* is our *horizontal resolution* and *n* is the line we're on. We need *w x h* bits to represent our screen spatially, where *w* is as before the *horizontal size* and *h* is the *vertical size* (both of them measured in *pixels*).

This is very straightforward and indeed the ZX Spectrum Next uses this way to store graphic data on *Layers 2*[2] and *Layer 1,0*. However in all the older modes, it uses a variation of linear storage called *interleaved* storage. The screen area is separated vertically into three 64 *pixel* high strips (or 8 *attribute cells*) arranged in blocks of 32. Each complete line (*x*) is stored linearly; in other words a *pixel* stored in horizontal coordinate 9 follows the *pixel* stored in horizontal coordinate 8 however, when it comes to the vertical order, there is a virtual hopscotch of sorts happening: The computer stores the first line of the first block of *attribute cells*, then stores the first line of the second block until it reaches the first line of the 8th block, then returns to the second line of the first block and the order continues with all second lines, then thirds and so on, until each third of the screen is full. *Fig.19* demonstrates the order of storage for ZX Spectrum Next legacy modes in order to visualise it a little better. We will get into more detail on why the graphic data is stored in that way later.

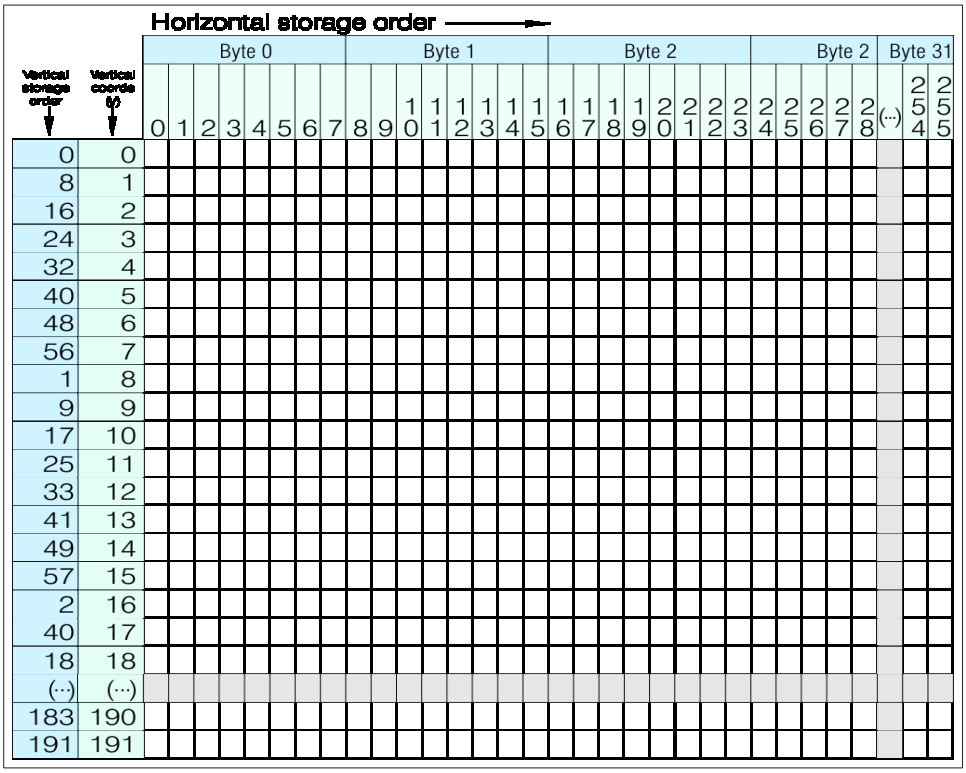



*Fig. 11 – Interleaved graphic data storage for ZX Spectrum Next standard resolution Legacy modes*

2 *Layer 2 higher resolutions (not currently supported by NextBASIC) store things a bit differently namely in 5 vertical strips of 16K each*

It's perhaps easier to understand the way things are stored by executing the following program:

```
10 LAYER 1,2
20 BANK 5 ERASE 0,6144,0
30 FOR %m=0 TO 6143
40 BANK 5 POKE %m,%@10101010
50 NEXT %m
```

This program will create vertical lines 1 *pixel* apart on your screen but will do so *in the order they are stored in memory*. As we saw previously **POKE** (and **BANK** *x* **POKE** *address*, *value*) writes a byte in memory at a specific address. The addresses we see starting with line **30** is where the *screen memory* is located and writing anything there will produce an image on your screen. The specific address 0 in BANK 5, marks a location called DISPLAY_FILE (or –alternatively– DISP_FILE1 but you'll see below why). It's important to note here that DISPLAY_FILE when dealing with legacy modes is always located at the same address: Byte 0 (decimal) or 0x0000 (hexadecimal) in BANK 5 (See *Chapter 23 – The Memory* for more details on the **BANK** command and its parameters).

*Layer 3* differs even more on how it stores data in memory. If you recall from *Chapter 14*, *Layer 3* is a *Character Graphics mode* and that name describes rather descriptively how it's arranged, in other words, very much like the screen is for regular **PRINT** commands as we saw in *Chapter 14*. The screen area is broken down to *rows* and *columns* and each of these locations, as marked by the unique *row by column coordinate*, points to a linearly stored 8 x 8 pixel image in memory called a *tile*. You can have up to *512* individual *tiles* in memory but you an also have as little as 1! Also the order of the *tiles* in memory is not important as each location can point to any *tile* from the ones available. In essence you can have an entire image composed of the same tile repeated over and over again much like you can fill a screen with "**A**" if you repeat a **PRINT "A"**; enough times. *Layer 3* therefore is an *array of pointers* to the *tile* locations in memory. One would ask, why is this complicated mechanism necessary? The answer is quite simple and you will see it repeated further down: By using pointers (in effect indices), we can translate much larger memory structures and requirements into simpler ones, ones that an 8-bit computer like the ZX Spectrum Next can manipulate easily. We will examine *Layer 3*'s memory organisation and usage separately and more in depth, at the end of this chapter and in the following two.

For all layers except *Layer 3,* the high resolution modes of *Layer 2* and the *Sprites Layer*, the ZX Spectrum Next has a maximum *horizontal resolution* of 512 *pixels*[3] and a *vertical resolution* of 192 pixels which gives us: 512 x 192 = 98304 *pixels* – or bits – in total or 12288 bytes. In order to store that, the ZX Spectrum Next defines a second DISPLAY_FILE area called DISP_FILE2 which is located at byte **8192** (decimal) or **2000h** (hexadecimal) in BANK 5. This secondary area has the same organisation as the first DISPLAY_FILE but when in use it holds the display of all odd-numbered *horizontal resolution* addresses letting DISP_FILE1 handle the even ones.

To demonstrate this visually you will need to edit the program above as follows:

```
10 LAYER 1,2
20 BANK 5 ERASE 0,6144,0
30 BANK 5 ERASE 8192,6144,0
40 FOR %m=0 TO 6143
50 BANK 5 POKE %m,%@10001000
60 NEXT %m
70 FOR %x=8192 TO 8192+6143
```

3 *The max horizontal resolution of 512 pixels is achieved by using half-width pixels which occupy the same area as the normal horizontal 256 full-width pixels.*

```
 80 BANK 5 POKE %x,%00100010
 90 NEXT %x
100 LAYER 0
```

then execute the program. The two LAYER statements first enable *HiRes* mode and then disable it. The two BANK 5 ERASE statements make sure there are no left over data in the DISP_FILE areas by filling them with 0s. You will see first the DISP_FILE1 area filling up and once the entire height of the screen is ran through, the DISP_FILE2 area doing the same. If you want to see this in a more dramatic way, convert line 10 to read LAYER 1,1 and then insert a line:

```
65 LAYER 1,2
```

This will illustrate even more vividly how the display is changed to handle odd and even *horizontal coordinates* from different areas of the memory.

So far, we learned that bits can have two states; 0 and 1; we are ready therefore to make the logical jump and assign two colour states for the image we just created. With 0 being black and 1 being white, we just defined a monochrome picture. But what about more colours?

We saw that we can display at least two colours on screen using a single bit. To display more (and store this information somewhere) we need to store more bits of information, with this information dealing exclusively with colour. In the beginning of this chapter we discussed how the ZX Spectrum Next generates and stores colour in 9 bits. The immediately obvious way to do that, would be to expand on the model displayed on *Fig. 18* by adding bits in the order the ZX Spectrum Next stores them and have a linear map of 9 bits per pixel. This is a good idea but unfortunately incorrect, and the reason for that goes back to our initial discussion of the ZX Spectrum Next being an 8-bit computer making accessing 9 bits of information at a time, extremely slow and therefore impractical in terms of design, both from software and hardware standpoints.

Instead the ZX Spectrum Next uses three systems of storing and displaying colour information additionally to the *HiRes* mode (*Layer 1,2*) which we just demonstrated as the latter is monochrome so no additional colour information is needed. These are:

1. Colour attribute display
2. Extended colour attribute display
3. Palette-based hybrid linear bitmapped colour display

## Colour attribute display

This system dates from the early ZX Spectrum models and was mainly conceived to both display colour and save on memory which at the time came at a premium. The graphic display is separated in 2 areas. The first which we already showed in the previous section (DISPLAY_FILE) only holds the actual 1-bit graphic data. Size-wise and for the standard resolution of *Layer 0* and *Layer 1,1*, this works out to: 256 x 192 = 49152 *pixels* – or – bits which divided by 8 gives us 6144 bytes which in turn divided by 1024 gives the 6 Kbytes figure). The second area, to which we shall introduce you now, is a smaller-sized memory block, known as COLOUR_FILE (or, alternatively, COL_FILE1) which resides immediately after DISPLAY_FILE in memory. It is 768 bytes long, and breaks down the colour information in blocks of 8 by 8 *pixels* (therefore dividing the screen in 32 x 24 blocks) or *attribute cells* where every cell can have two possible colours out of a total of 8 simultaneously. This colour information is stored in two consecutive GRB blocks of three bits each, preambled

by two additional bits that can make the colours flashing and/or brighter. *Fig. 12,* shows how colour information is stored in each byte in the COLOUR_FILE area.

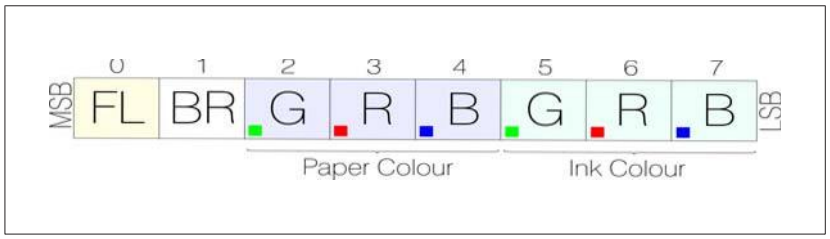



*Fig. 12 - Attribute byte organisation*

The two colours stored within are named INK and PAPER mainly to reference the printed characters we explored in the previous chapter since INK is the colour of the character itself and PAPER is the rest of the background, in a sense a form of virtual paper we write on[4]. That way *Layer 0* graphics can display up to 16 colours on screen using very little memory but with the tradeoff of *colour clash*. This term simply describes the fact that the *colour resolution* is much lower than the *spatial* one.

Like its DISPLAY_FILE counterpart, COLOUR_FILE can have a secondary area which, when enabled, is called COL_FILE2 and resides right after DISP_FILE2.

Unlike the DISPLAY_FILE areas, COLOUR_FILE areas are *normally* straightforward in how they are stored and that is simply in order of cells from top leftmost to right bottommost.

The secondary DISPLAY_FILE area, other than the *HiRes* (*Layer 1,2*) area for even display addresses can also function as a *shadow screen* which is a non-visible screen, identical in organisation to the first one, that holds a visual we may want to project quickly thus creating animation effects as we'll see in *Chapter 17 – Time and Motion* later on. In that usage the secondary COLOUR_FILE area functions exactly the same way as the primary one. In *HiColour* mode however (*Layer 1,3*), DISP_FILE2 becomes itself a COLOUR_FILE and the normal COL_FILE1 and COL_FILE2 are not used. It's also noteworthy, that *HiRes* mode also does not use the COLOUR_FILE areas but for a different reason

*HiColour* mode (*Layer 1,3*) reduces the amount of *colour clash* by reducing the size of *attribute cells* thereby extending the colour resolution to 32x192 cells of 8x1 *pixels* in size.

As the colour resolution increases, the memory requirements are increased as well and that is why the entire memory of DISPLAY_FILE2 is used in lieu of a COLOUR_FILE. It's easy to figure out why this happens: The original COLOUR_FILE area of 768 bytes is extended (therefore multiplied) by 8 times to make the vertical colour resolution equal to the spatial resolution. If you make the multiplication 768 x 8 you see that a further 6.144 bytes are needed to increase the colour resolution. COLOUR_FILE1 and COLOUR_FILE2 areas are unused in this mode. The organisation however of this enlarged COLOUR_FILE since the colour resolution has grown follows the one of the DISPLAY_FILE meaning that it uses the same interleaved storage as the graphic data.

We can therefore modify our original program to also display colour attributes so we can get a visual idea of the two modes' differences:

```
10 LAYER 1,1
20 BANK 5 ERASE 0,6912,4
30 BANK 5 ERASE 8192,6912,4
40 FOR %m=0 TO 6143
50 BANK 5 POKE %m,%@10101010
60 NEXT %m
```

4 *This distinction is purely arbitrary but it helps distinguish these two colours from one another in a more human–readable way. They could have been easily called COLOUR_A and COLOUR_B.*

```
 70 FOR %a=6144 TO 6144+767
 80 BANK 5 POKE
    %a,INT((RND*1)+0.2)*128 +
    %RND(128)
 90 NEXT %a
100 LAYER 1,3
110 FOR %x=8192 TO 8192+6143
120 BANK 5 POKE
    %x,INT((RND*1)+0.2)*128
    + %RND(128)
130 NEXT %x
140 LAYER 1,1
150 PAUSE 0
```

Lines 20 and 30 clear the DISP_FILE1 and DISP_FILE2 memory, Lines 70 to 90 fill the COL_FILE1 area with random colour information. The LAYER 1,3 command in Line 100 switches to *HiColour* mode and subsequently random colour information is written in each *attribute cell* with lines 110 to 130. As you can see, attribute cells in *HiColour* mode are much smaller in size and written in an *interleaved* manner as opposed to the *linear* manner demonstrated by lines 70 to 90. Finally line 140 switches back to *Layer 1,1*. To increase the variety of colour combinations and reduce the times of flashing being introduced the FLASH bit is randomised independently.

## Extended colour attribute display

The creation of the ZX Spectrum Next brought forth *Layer 2* and its extended resolutions and colours. However the need for colourisation of older software arose. What could be done to give a part of the new features to older software without breaking compatibility or having to rewrite from scratch? There have been many solutions offered since the inception of the original ZX Spectrum, each with its own strengths and drawbacks but all had been difficult, and most non-accessible in a straight forward manner from BASIC. A solution in the form of an *EnhancedULA* was conceived therefore that would give access to the entirety of the ZX Spectrum Next's colour capability without sacrificing compatibility or ease of use.

This is achieved by retaining the DISPLAY_FILE and COLOUR_FILE memory areas but rearranging COLOUR_FILE byte organisation by repurposing the FLASH and BRIGHT bits and increasing the amount of INK and PAPER bits which become pointers to *palette* colours (see the following sections for more information on *palettes*). This way, simple commands allow recolouring of older software which is not aware of the ZX Spectrum Next's colour 'abilities' without sacrificing compatibility. *Colour clash* remains (as do the *attribute cell* sizes) however the colour capabilities extend to a maximum of 256 colours out of the 512 the ZX Spectrum Next can display. To demonstrate (without getting into too much detail) how you can use more colours using the *Extended colour attributes display* of the *EnhancedULA* type the following program:

```
10 BANK NEW ba
20 FOR %a=0 TO 255
30 BANK ba POKE %a,%a
40 NEXT %a
50 LAYER 1,1
60 PALETTE DIM 8
70 LAYER PALETTE 0 BANK ba, 0
```

```
 80 PALETTE FORMAT 255
 90 BANK 5 ERASE 0,6912,255
100 %l=6144
110 REPEAT: WHILE %l<6912
120 IF %c>255 THEN %c=0
130 BANK 5 POKE %l,%c
140 %l,%c+=%1
160 REPEAT UNTIL 0
170 PAUSE 0
```

Don't worry about the unknown commands yet. What the program does is to create an 8-bit palette for Layer 1,1, then enable the EnhancedULA and switch it to *Full Ink Mode* then cycle through all 256 colours of that palette by writing in the COLOUR_FILE area the specific attribute. We'll go into more detail on how that works when we examine IN and OUT and the ZX Spectrum Next Ports System in *Chapter 22*

## Palette-based hybrid linear bitmapped colour display

This system of colour organisation, storage and display is applicable to *Layer 1,0*, *Layer 2*, *Layer 3* and partly applicable to the *Sprite System*. Before we explain why it's hybrid, we'll point you back to the beginning of this chapter and especially the *Colour organisation and representation* section. As you recall, we said there that the ZX Spectrum Next can handle both 9-bit and 8-bit colour. This is technically inaccurate as we have a broader spectrum of colours that a single byte can display.

We'll take a small detour here and explain the concept of a palette. A palette is a subset of colours where each colour displayed on screen, is not actually stored as the colour component information it's made up of, but rather as a pointer (or index) of the actual colour that's stored somewhere else. This subset in the ZX Spectrum Next's case is comprised of either 256 pointers (therefore we require only an 8-bit number to store each pointer) or 16 pointers plus one offset (therefore we require only a 4-bit number to store each pointer with an additional 4-bit number to point us to one of 16 groups of colours) to the actual colours which are represented by 16-bit numbers (therefore a set of *two* 8-bit consecutive numbers which have 6 bits[5] unused, give the 9 bits of the actual colour stored, albeit rather inefficiently).

There are 8 palettes in the ZX Spectrum Next. Two for each graphics system:

- Layers 0 and 1 use two
- Layer 2 uses two more
- Layer 3 also uses two –and–
- The Sprite System uses the last two

With two palettes, all 512 possible colours of the ZX Spectrum Next can be recalled, rearranged and stored and therefore assigned to pixels, character tiles or attribute cells according to the layer in use, on screen. That doesn't mean all can be displayed simultaneously without some clever *NextBASIC tricks*. Normally only 256 can be shown on a particular layer at one time.

The ZX Spectrum Next palette system has a special mode where if one were to use an 8-bit colour in the R3G3B2 format and assign one palette in sequence to the value that equals the pointer value (for example set palette location 15 to be of a value 15) then we could treat the entire display of *Layers 2* and *Layer 1,0* (*LoRes*) as 8-bit, treating from then on the display instead of a palette-based one, as a bitmapped one. This is exactly why we can call it hybrid.

5 *Obviously 16 positions minus 9 positions should equal 7 unused positions, however there's one more bit used called the 'priority bit' which although unused in the case of other layers, is used in Layer 2 palettes as we'll see in the next section.*

In reality, each R3G3B2 colour is translated internally by the ZX Spectrum Next into a full RGB3 colour by performing a binary *OR* of the first bit (MSB) of the blue component with 0 so for example colour 10111110 (8-bit) will become internally 101111101 as a full 9-bit colour.

*Layer 1,0* (*LoRes) and Layer 2* are very straightforward in how they store both colour as well as graphic data. Unlike the other modes, there's no separate area for colour and there is no interleaving in the order of storage or separate pointers to the area the data is stored. Each byte of memory represents one pixel on screen from the top left to the bottom right. The only two differences between them are the memory location where the screen contents are stored and their resolution. The former uses the standard DISP_FILE1 and DISP_FILE2 areas (each holding one half of the screen) and is usually stored in BANK 5, having a maximum resolution of 128 w x 96 h pixels (thus making it a total of 12 Kbytes in size) while the latter uses up to 5 banks. The default resolution of 256 w x 192 h pixels uses normally 3 (by default BANKS 9,10 and 11 but these are *relocatable*[6]), making its memory requirements 48 Kbytes while the higher resolutions of 320 w x 256 h and 640 w x 256 h use 5 16 Kbyte banks, requiring a total of 80 Kbytes.

Although the remaining, higher resolution, *Layer 2* modes are not currently supported by *NextBASIC* (they will however in the near future) it is important to mention them here for two reasons:

First, because they both store their data (as we will see in the last section of the following chapter) in a sideways manner and secondly because especially for the highest resolution *Layer 2* mode (640 w x 256 h) the colour is stored in a similar manner as *Layer 3* below.

That being said, in all *Layer 2* resolutions, colour is stored as part of the graphic data and therefore neither *LoRes* nor *Layer 2* modes suffer from colour clash.

An additional side-effect of the linear nature of these modes, is that the concepts of FLASH and BRIGHT do not exist there. BRIGHT was just a way to squeeze more colours out of a very limited selection and FLASH can be reproduced by quickly inverting the contents of an area using a number of programming techniques available via *NextBASIC*.

## Layer 3 colour storage

*Layer 3* is special as it allows for complete usage of the full Spectrum Next screen area, therefore the entirety of the 320 w x 256 h pixels resolution is available (combining the standard graphic area with the width and height of the border) and uses either (like the Sprite system we will examine in *Chapter 17*) a *palette offset + 4-bit index* combination to store colour for each tile or a monochrome mode specifically suited to display text. The first method, achieves significant memory space savings without sacrificing colour capabilities (although at first it may look a bit restrictive): Each tile being 8 x 8 has 64 possible pixel locations; by using a 4-bit colour index number we can only have $2^4 = 16$ combinations/colours instead of 64 we theoretically could have. With a bit of prior arrangement of our image data however, we can achieve spectacular results and display very complex images (colour-wise) even with that restriction in place. The monochrome mode has obvious memory benefits we have explored with *Layer 1,2* (HiRes) as well as increased speed.

## Layer 2 priority colours

As we will see in length on the following section, since the ZX Spectrum Next display is *layered,* there is a way to rearrange the layer display priority, or rather the order in which these layers are stacked one on top of another. This provides unique flexibility however there are cases that you'd want to mesh the layers in a more complex way as for example in the case of a game where you would want the player's *sprite* to weave in-and-out the environment in order to get the impression of depth. Usually this is achieved by employing an algorithm that performs *environmental masking*; hiding in other words things that we don't

6 *By relocatable, we mean that although the ZX Spectrum Next initially reserves BANKS 9 through 12 for Layer 2 graphic data, this can change either automatically or by the user. One should not assume the aforementioned banks of memory always hold Layer 2 graphic data. Check Chapters 22 and 23 for more information regarding the actual Layer 2 location.*

want to display on the top layer. This process, especially where it involves moving graphics, is very *processor-intensive* and can slow down the computer, resulting in a not-so-fluid experience of movement. The ZX Spectrum Next addresses this very specific issue with the introduction of *priority colours*. These apply only to *Layer 2* palettes and are defined by setting the *8th* bit of the *secondary byte* of each palette entry to **1**.

Setting any palette entry's *priority bit* will ensure that this colour will always print *on top of everything else*. In case you would need the same colour to exist in a layer below the top-most you will need to define the same colour again but on a different index using the LAYER PALETTE command.

We will revisit this topic further below, when we reach the palette manipulation commands.

## More on the LAYER command

In *Chapter 14* as well as in the previous sections of this chapter we saw repeated mentions and usage of the LAYER command. By now, you should have enough grasp of the mechanics behind the ZX Spectrum Next's colour and graphic system to examine it in a little more detail. We will further expand on its usage every time a functionality we haven't yet discussed is introduced (as in the PALETTE section that follows shortly) but for now let's head back to the beginning of *Chapter 14* and re-iterate the possible graphic modes in conjunction with LAYER which is used to change between them.

First of all and given what we've learned in terms of colour, it's helpful to conceptualise the graphic system in a slightly different manner than what the LAYER command organises them in. These layers are grouped together in terms of functionality and memory addresses they use, namely: The *ULA* modes (*Layer 0 and all Layer 1* modes), *Layer 2* and the *Sprite System* (which we will examine in more detail in *Chapter 17 – Time and Motion*). This can get a bit confusing as *LoRes (Layer 1,0)* and *Layer 2* use the same colour storage and display system so it's better to completely disregard this and instead imagine four different screens laying on top of one another with programmable priorities and potential transparency. In simple words that means that you can select whichever screen you want to appear on top and in which order. This means putting a priority onto the memory space that holds the data for the graphics and displaying this above everything else. This is achieved with the

LAYER OVER *order*

command, where order is one of the following:

0 Sprites over Layer 2 over ULA (Layer 1) – the default
1 Layer 2 over Sprites over ULA (Layer 1)
2 Sprites over ULA (Layer 1) over Layer 2
3 Layer 2 over ULA (Layer 1) over Sprites
4 ULA (Layer 1) over Sprites over Layer 2
5 ULA (Layer 1) over Layer 2 over Sprites
6 Sprites over (Layer 2 + ULA combined) – colours clamped to 7
7 Sprites over (Layer 2 + ULA combined) – colours clamped to (0,7)

The last two ordinals enable one of the two colour blending modes allowing for some very interesting lighting/shading effects.

This (as we will see in *Chapter 22 – IN, OUT and the Next Registers*) directly affects the *Sprite and Layer System Register* (Register 21) and in the same order as the LAYER OVER command.

*Fig. 13 below visualises the way layers compound, to form the ZX Spectrum Next display.*

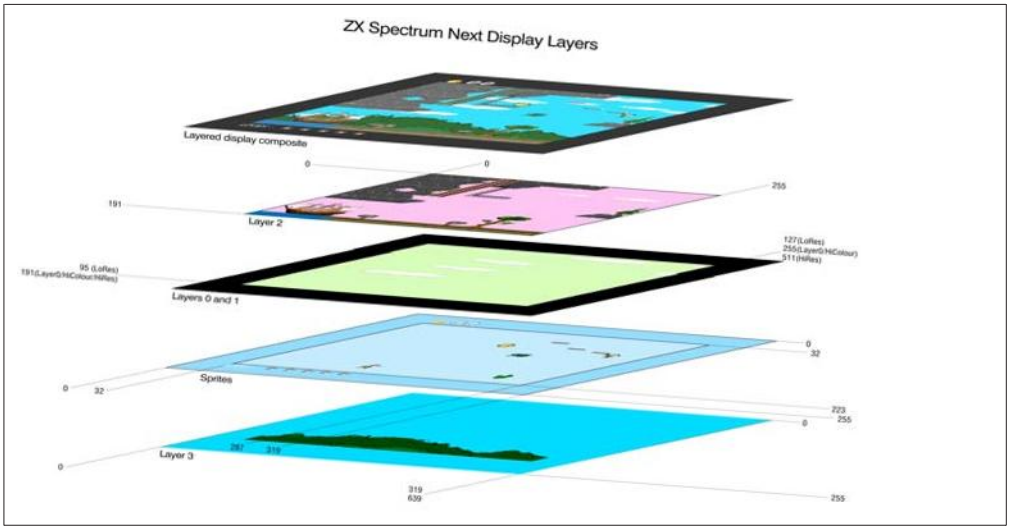



*Fig. 13 – Display Layers*
*(Graphics courtesy of Lampros Potamianos from: The Hollow Earth Hypothesis)*

You will notice a few odd things about the diagram above. First, it is out of order with the sprites appearing below Layers 0 – 2. That brings us to the second thing (don't worry the dots will be connected shortly) which is that the *Sprite Layer* as well as *Layer 3* have a higher usable resolution than Layers 0 through 2. The order was changed to group the like resolutions ranges together and better visualise that Layer 3 as well as the Sprite System have a maximum of 320 pixel horizontal by 256 pixels vertical resolution as opposed to the 256 pixel by 192 pixel standard pixel size resolution of the other layers[7]. As for the order as seen in the LAYER OVER command, it really doesn't matter, as it can be rearranged in the way we see fit. In the specific example above we can see how one can mix-and-match several Layers to construct a more complex final visual; *Layer 3* is used for the background, the extended sprite area for relatively static information about the game (Lives and score), *LoRes* (Layer 1,0) for basic parallax animation (clouds) and *Layer 2* for the remaining more complex and colourful graphics.

It's also noteworthy, that although we spoke about memory organisation in regards to colour for all layers, we did not do so for the *Sprite System*. That is because sprites do not occupy normal memory but instead, use their own dedicated memory that's located within the *Next Sprite Engine* hardware. The LAYER command other than to set priorities of display does not affect, nor addresses the *Sprite Engine* directly therefore in the following commands, the latter is not referenced anywhere.

There are more LAYER compound commands that are more pertinent to graphics rather than colour and others that deal with motion in some fashion or other. We will revisit therefore LAYER in more detail in the following sections and chapters. The main functionality of the LAYER command which is none other than changing graphic modes.

LAYER *number*, *parameter*

will change the *layer* to the one specified by *number* with an optional *parameter* according to the list below:

| | |
|---|---|
| LAYER 0 | Select legacy ZX Spectrum Mode |
| LAYER 1,0 | Select Layer 1, LoRes mode |
| LAYER 1,1 | Select Layer 1, standard resolution mode |
| LAYER 1,2 | Select Layer 1, HiRes mode[8] |

7 *Layer 2 supports higher resolutions but as far as NextBASIC is concerned, currently only 256 x 192 is usable*
8 *HiColour and HiRes modes are also called Timex modes as they were originally introduced in the Timex Sinclair TS2068 advanced ZX Spectrum compatible computer which was released primarily for the US market in 1983.*

| | |
|---|---|
| LAYER 1,3 | Select Layer 1, HiColour mode |
| LAYER 2 | Select Layer 2 mode |
| LAYER 2,0 | Select Layer 2 mode and disable its display |
| LAYER 2,1 | Select Layer 2 mode and enable its display |

Attempting to enter a layer number or parameter that's not supported according to this list, will result to a B Integer out of range error.

There's one more command of note and this is:

### LAYER CLEAR

which will reset all layer information, including banks, mode, the Layer 2 display enable, layer offsets (see *Chapter 17*) and ordering to defaults. This is also done by NEW.

## BORDER, PAPER, INK, BRIGHT and FLASH

Run this program:

```
  5 LAYER 0
 10 FOR m=0 TO 1: BRIGHT m
 20 FOR n=1 TO 10
 30 FOR c=0 TO 7
 40 PAPER c: PRINT "    ";;; 4 spaces
 50 NEXT c: NEXT n: NEXT m
 60 FOR m=0 TO 1: BRIGHT m: PAPER 7
 70 FOR c=0 TO 3
 80 INK c: PRINT c;"   ";;; 3 spaces
 90 NEXT c: PAPER 0
100 FOR c=4 TO 7
110 INK c: PRINT c;"   ";;;3 spaces
120 NEXT c: NEXT m
130 PAPER 7: INK 0: BRIGHT 0
```

This shows the fifteen colours (including white and black and the BRIGHT variants) that the ZX Spectrum Next can produce on the screen if switched to *Layer 0* (or standard resolution modes of *Layer 1*) without the *EnhancedULA* functions enabled. Here is a list of the basic eight for reference; they are also written over the appropriate number keys on your ZX Spectrum Next's keyboard:

| | |
|---|---|
| 0 | black |
| 1 | blue |
| 2 | red |
| 3 | purple –or magenta– |
| 4 | green |
| 5 | cyan –or pale blue– |
| 6 | yellow |
| 7 | white |

If you're thinking to yourself that the total colours (taking account of brightness turned on) should be 16, you'd be technically right however there cannot be a BRIGHT black so the total amount of colours is indeed 15. As you've noticed, the program introduces three commands: PAPER, INK and BRIGHT. If you look back to the *Colour attribute display* section you will recognize the terms immediately. These commands are the primary way of applying colour to objects on screen in *NextBASIC*. There is a number of supporting colour commands as well which will examine further in the following sections.

Before we delve a bit deeper into what each does and how, it's very important to understand that the commands operate differently according to the *layer* we're on and this points back to the different way the ZX Spectrum Next stores colour. When we're dealing with modes that make use of *attribute cells*, we need to think in terms of those cells. PAPER there affects the background or, in other words, the place in the cell where graphic data is *non existent* (set to 0) whereas INK does the exact opposite and affects areas within the same cell where graphic data is *existent* (set to 1). Moreover these commands affect the entire *attribute cell* and not just one singular pixel within the cell. In other words, it doesn't matter how many times you set the INK or PAPER within a particular cell, only the *last command will be the one that has the permanent effect* for that cell. BRIGHT similarly affects the entire cell as we already saw, however it does absolutely nothing if *EnhancedULA* is enabled or if we are on modes that do not support attributes like *HiRes*, *LoRes* and *Layer 2*.

On *LoRes* and *Layer2*, since *attribute cells* do not exist, the entire notion of PAPER and INK should be irrelevant. It is easier, however, for the user to understand them in similar terms as the *attribute display* modes i.e. in terms of a *character-based* display. Indeed, there's nothing stopping us from having an 8 x 8 character drawn on screen (say a 2) with every single *pixel* around the character having a different colour, something that's impossible on *attribute display* modes. This however would be very difficult to do in terms of a singular colour command and for that reason PAPER and INK commands were simply extended to work in a similar manner as their *attribute cell* modes' counterparts even where their underlying mechanics are different. On the other hand, in *HiRes* mode, PAPER and INK commands only serve the purpose of selecting a colour scheme as we will see below. The following table shows all primary colour commands functionality according to the graphics mode we're in.

| | Attribute Modes | | | | Non-Attribute Modes | | |
|---|---|---|---|---|---|---|---|
| | Standard ULA | | | EnhancedULA | | | |
| | Layer 0 | Layer 1,1 | HiColour | Layer 1,1, HiColour | HiRes | LoRes | Layer2 |
| INK | 0-9*** | 0-7 | | 0-255 | 0-7* | 0-255 | 0-255 |
| PAPER | 0-9*** | 0-7 | | 0-255 | 0-7* | 0-255 | 0-255 |
| BORDER | 0-7 | | | 0-7 | 0-7** | 0-7 | 0-7 |
| FLASH | 0-1,8*** | 0-1 | | N/A | N/A | N/A | N/A |
| BRIGHT | 0-1,8*** | 0-1 | | N/A | N/A | N/A | N/A |
| Palette in use | ULA | | | ULA | ULA | ULA | L2 |

* INK in HiRes is complimentary to PAPER i.e. when INK is 0 then PAPER is 7 and if INK is 3 then PAPER is 4 and so on.
** BORDER has no effect but it's set by the PAPER setting
*** INK/PAPER/BRIGHT/FLASH 8 mean Transparent, ergo it preserves the colour setting that was there previously and INK/PAPER 9 mean Contrast, ie. the complimentary colour of the other statement (something similar to PAPER/INK settings for HiRes modes)

*Table 7 – Colour commands' functionality according to Graphics Mode/Layer*

There is another way of using INK, PAPER etc, which you will probably find more useful than having them as statements. You can put them as items in a PRINT statement (followed by ;), and they then do exactly the same as they would have done if they had been used as statements on their own, except that their effect is only temporary: it lasts as far as the end of the PRINT statement that contains them. Thus if you type:

```
PRINT PAPER 6;"x";: PRINT "y"
```

then only the x will be on a yellow background.

When used as statements in *Layer 0*, INK, PAPER, BRIGHT and FLASH, do not affect the colours of the lower part of the screen, where commands and INPUT data are typed in. The lower part of the screen uses the colour of the BORDER as its PAPER colour, value 9 for contrast as its INK colour, has FLASH turned off, and everything is set at normal BRIGHT.

## BORDER

Undoubtedly, you have noticed thus far, that there is an area you cannot write –normally– to, surrounding the area where you can print or draw graphics over. This area is called the BORDER and using standard *NextBASIC* statements you can only change its colour. The statement:

BORDER colour

changes the *border* colour to any of the eight normal colours (not 8 or 9) or colours changed by the PALETTE statement we shall explore below in length.

## INVERSE and OVER

There are two more statements, INVERSE and OVER, which, when in an attribute mode, control not the attributes, but the actual graphic data that is printed on the screen. They use the numbers 0 for *off* and 1 for *on* in the same way as FLASH and BRIGHT do, but those are the only possibilities. If you do INVERSE 1, then the graphic data printed will be the inverse of their usual form: paper *pixels* will be replaced by ink *pixels* and vice versa.

The statement:

OVER 1

sets into action a particular sort of overprinting. Normally when something is written into a character position it completely obliterates what was there before; but now the new character will simply be added in on top of the old one (but see *Exercise 1*). Note that if the character you're overprinting with has a pixel in the same position with the character you're printing OVER, the result will be a blank pixel. In other words, OVER is a *XOR* operation.

This can be particularly useful for writing composite characters, like letters with accents on them, as in this program to print out German letters – an o with an umlaut above it:

```
10 OVER 1
20 FOR n=1 TO 32
30 PRINT "o"; CHR$ 8;"""";
40 NEXT n
```

(notice the control character CHR$ 8 which backs up one space.)

## Using colour control codes

The previous example, reminded us of the PRINT positioning control codes. We can do exactly the same with colours by using the special colour control codes in a similar manner like the one we explored in *Chapter 14*.

The colour control codes are:

CHR$ 16 corresponds to INK
CHR$ 17 corresponds to PAPER
CHR$ 18 corresponds to FLASH
CHR$ 19 corresponds to BRIGHT
CHR$ 20 corresponds to INVERSE
CHR$ 21 corresponds to OVER

These are each followed by one character that shows a colour by its code: so (for instance):

```
PRINT CHR$ 16 + CHR$ 9; …
```

has the same effect as:

```
PRINT INK 9; ...
```

## ATTR

The ATTR function has the form:

ATTR (*line*, *column*)

Its two arguments are the *line* and *column* numbers that you would use in an AT item, and its result is a number that shows the colours and so on at the corresponding character position on the screen. You can use this as freely in expressions as you can any other function.
The number that is the result is the sum of four other numbers as follows:

128 if the character position is flashing, 0 if it is steady
64 if the character position is bright, 0 if it is normal
8 *times* the code for the paper colour –and finally–
the code for the ink colour

For instance, if the character position is flashing and normal with yellow paper and blue ink then the four numbers that we have to add together are 128,0,8*6=48 and 1, making 177 altogether. Test this with:

```
PRINT AT 0,0; FLASH 1; PAPER 6; INK 1;
" "; ATTR (0,0)
```

ATTR works only on *Layer 0* and that is because it works by reading each COLOUR_FILE location. On different modes where the memory organisation and usage differs it will return a number that corresponds to the original COLOUR_FILE memory location, which could be for all purposes nonsense. That being said, you can get information on the extended colour attribute display if the *EnhancedULA* functions are enabled presuming the screen area hasn't moved. That number will correspond to the indices in use and it changes according to which PALETTE FORMAT command is in effect as we'll see below. For other modes it's safer to use the POINT TO command which we will examine in *Chapter 16.*

## PALETTE

In previous sections of this chapter we got introduced to the subject of palettes and how they affect colour display and manipulation in each of the colour modes. We also got briefly introduced to the PALETTE keyword and a few of its uses. We can now expand a bit more on the subject, as PALETTE not only affects printing of the characters on screen but also all aspects of graphics including the ZX Spectrum Next's Sprite Engine.

The PALETTE keyword can be used as a primary statement or as a modifier to the LAYER and SPRITE statements to perform a variety of functions that pertain to colour manipulation.

As we saw, colour on the ZX Spectrum Next when using *extended colour attribute display* or any mode that doesn't use attributes, can be defined using 9 bits or 8 bits per colour. The default is 9; when 8 bits are chosen, as we have already seen previously, non attribute modes can emulate a straight-up bitmapped linear display (with the side-effect that only 4 levels of blue are available). In the latter case you can basically ignore all PALETTE statements as non-applicable for *Layer 2* –and this whole section for that matter– however you need to use them if you want to manipulate *LoRes* or any of the *Layer 1* and *Layer 0* modes and/or change the default colours anywhere in your system, or even to recolour an old game. In order to do that and to have access to the broadest gamut of colour you will need to change the *bit-depth* of your palette(s). You can do so with the PALETTE DIM statement in the form:

PALETTE DIM *bits*

where *bits* can be 8 or 9.

The default colour mode of *Layer 1* modes (except *LoRes* and *HiRes*) is the standard *colour attribute display* one. In order to enable the *extended colour attribute display* mode we need to enable the *EnhancedULA* functionality. For this you must use the PALETTE FORMAT which takes the form:

PALETTE FORMAT *ink_count*

where *ink_count* is a numerical expression specifying the number of inks to be in the palette (0,1,3,7,15,31,63,127 or 255). When the *EnhancedULA* is enabled, BRIGHT and FLASH are ignored, and INK and PAPER accept the appropriate new range of values. Note here that although you can specify INK and PAPER values up to 255 when writing a program, attempting to execute the program in Layer 0 will result into a K Invalid Colour error when the *EnhancedULA* is not enabled. To disable the *EnhancedULA* functionality you will need to specify an ink count of 0. The standard attributes with 8 inks, 8 papers, bright and flash are then once again supported.

As we saw in *Fig. 13* there is an order of display of different layers on screen. Although it is not immediately apparent this means that it's also possible to mix display output from more than one graphical layers. That is achieved by assigning a *global transparency mask* for the regular layers or, in the case of the *Sprites* layer, a *transparency index*, and then colouring the areas or sprites we want to be transparent with the specific colour.

You can set the *transparency colour mask* or *transparency colour index* using the following statement:

PALETTE OVER *value*

where *value* is an 8-bit numeric expression which identifies a colour either in R3G3B2 8-bit format (in the case of regular graphics layers) or the index to the 9bit colour value we want to be transparent (in the case of the *Sprites* layer). The default *global transparency mask* and *transparency colour index* is light magenta / 227 (11100011 in binary).

To reset all palette data and settings to default, use the PALETTE CLEAR statement.

In the *Palette-based hybrid linear bitmapped colour display* section, we first discussed the existence of two palettes per display layer (note here that in this case layer is meant in the memory usage paradigm displayed in *Fig. 13* so *ULA layers* get grouped together).

We can switch between palettes using the compound keyword:

LAYER PALETTE *n*

where *n* is the palette to use (0 or 1) for the current *memory usage layer* (ie. if you're in any *ULA layer* all of it gets affected but not *Layer 2* etc).

You can point a palette for the current layer to palette data you have previously stored in memory using the following compound command:

LAYER PALETTE *number* BANK *bank*, *offset*

where *number* is the palette to update (0 or 1) for the current *memory usage layer*, *bank* is the memory bank to point to, and *offset* is the offset within that memory bank (For more information about BANK see *Chapter 23 – The Memory*).

Palette data should be either 256 double byte colour entries (for 9-bit), or 256 single byte entries (for 8-bit). As per what we discussed earlier in the chapter we need to encode the colour information in an R3G3B2 (for 8-bit) or RGB3 (for 9-bit) with every colour component value describing 8 intensities per colour.

In the double-byte entry method, the second byte in each sequence only has one bit defined for colour: the 3rd blue bit as well as one bit for priority (which only applies to palettes

used for *Layer2*)[9]. It may seem to be a bit inefficient as it stands, because it appears to be wasting memory but that's only if we store our palette in memory before we load it, otherwise palettes do not use memory at all and they only need to be set once and the memory used by the BANK method can be immediately released to the system.

You have already seen an example of this method in the *Extended Colour Attribute Display System* section where a palette is set up first as colours and then assigned into the chosen layer palette. Could you change it to accept double-byte colour values?

The tables that follow, show the proper format for single and double-byte palette entries. The integer values are included for a better understanding of the conversion process. In actuality, you can use either the BIN keyword or the %@ qualifier to enter binary numbers directly.

| First Byte | | | | | | | | Second Byte | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $R_1$ | $R_2$ | $R_3$ | $G_1$ | $G_2$ | $G_3$ | $B_1$ | $B_2$ | P2 | 0 | 0 | 0 | 0 | 0 | 0 | $B_3$ |
| 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 4 | 2 | 1 | 4 | 2 | 1 | 2 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

*Table 8– Double byte colour entry*

| First Byte | | | | | | | |
|---|---|---|---|---|---|---|---|
| $R_1$ | $R_2$ | $R_3$ | $G_1$ | $G_2$ | $G_3$ | $B_1$ | $B_2$ |
| 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
| 4 | 2 | 1 | 4 | 2 | 1 | 2 | 1 |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

*Table 9 – Single byte colour entry*

Writing the entire palette into memory is not the only option available to the user in order to program a palette. It is also possible to specify individual colours within the palette using the following compound command (as with the rest of the examples in this section layer here implies a memory space organisational unit):

LAYER PALETTE *number*, *index*, *value*

where *number* is the current layer palette we wish to update (0 or 1), *index* is the index of the palette entry to be updated (0 to 255), and *value* is the colour components value expressed in binary using either the BIN keyword or the %@ qualifier in RGB3 format. That means that the colour in that case is ALWAYS 9-bit For example:

```
LAYER PALETTE 0,0,BIN 110010011
```

that sets colour index 0 in palette 0 to a nice pink is exactly the same as:

```
LAYER PALETTE 0,0,%@110010011
```

## Exercises

1. Try:

```
PRINT "B"; CHR$ 8; OVER 1; "/"
```

Where the / has cut through the B, it has left a white dot. This is the way overprinting works on the ZX Spectrum: two papers or two inks give a paper, one of each gives an ink. This has the interesting property that if you overprint with the same thing twice you get back what you started off with. If you now type:

9 *In the future 15-bit (with priority) or even 16-bit (without priority) may be possible on an HDMI display as the hardware is very capable of displaying it albeit slowly*

```
PRINT CHR$ 8; OVER 1; "/"
```

Why do you recover an unblemished B?

2. Type:

```
PAPER 0: INK 0
```

Isn't it just as well that these don't affect the lower part of the screen?
Now type:

```
BORDER 0
```

and see how well the computer looks after you!
But what will happen if you do the same after giving:

```
LAYER 1,3
```

3. Run this program:

```
10  POKE 22527+RND*704, RND*127
20  GO TO 10
```

Never mind how this works; it is changing the colours of squares on the screen and the RNDs should ensure that this happens randomly. The diagonal stripes that you eventually see are a manifestation of the hidden pattern in RND – the pattern that makes it *pseudorandom* instead of truly random.

4. Type in the chess piece characters in *Chapter 13* and then type in this program which draws a diagram of chess positions using them:

```
5  REM draw blank board
10 bb,bw=1,2: REM red and
   blue for board
15 PAPER bw: INK bb: CLS
20 PLOT 79,128: REM border
30 DRAW 65,0: DRAW 0,-65
40 DRAW -65,0: DRAW 0,65
50 PAPER bb
60 REM board
70 FOR n=0 TO 3: FOR m=0 TO 3
80 PRINT AT 6+2*n, 11+2*m;" "
90 PRINT AT 7+2*n, 10+2*m;" "
100 NEXT m: NEXT n
110 PAPER 8
120 pw,pb=6,5: REM colours of
     white and black pieces
200 DIM b$(8,8): REM positions of pieces
205 REM set up initial positions
210 b$(1)="rnbqkbnr"
220 b$(2)="pppppppp"
230 b$(7)="PPPPPPPP"
```

```
240 b$(8)="RNBQKBNR"
300 REM display board
310 FOR n=1 TO 8: FOR m=1 TO 8
320 bc=CODE b$(n,m): INK pw
325 IF bc=CODE " " THEN GO TO 350
    : REM   space
330 IF bc>CODE "Z" THEN INK pb:
     bc-=32:;lowercase for
    black
340 bc+=79:;convert to
    graphics
350 PRINT AT 5+n, 9+m; CHR$ bc
360 NEXT m: NEXT n
400 PAPER 7: INK 0
```

5. The program in *p. 86* has a non-apparent flaw. Can you improve on it so it becomes faster?

6. Write a version of ATTR using a PROCedure that will work always, no matter the mode. You can peek ahead if you so wish!

7. Using the global transparency colour, palettes and layers can you write a program that will display ALL 512 colours of the ZX Spectrum Next on screen? (It's easier than you think)

# Chapter 16 – Graphics

In this chapter, we shall see how to draw pictures on your ZX Spectrum Next's screen. As we learned in *Chapters 14 and 15, Layer 0* can only use 175 pixels out of its maximum 192 pixel vertical resolution while the other layers accept the maximum height defined by the layer as their vertical resolution. Moreover, if you recall *Fig. 9* and *10*, *Layer 0* has a different graphics coordinate origin from the rest of the layers/modes located at the bottom leftmost of the screen instead of the top leftmost. All basic graphics commands that we will explore (PLOT, DRAW, CIRCLE and POINT) accept both coordinate origins while the LAYER and TILE commands (as well as the SPRITE command we'll explore in the following chapter) accept only the top leftmost corner as the coordinate origin. The side-effect of these inverted coordinate systems is that most graphics you will program will appear inverted on the y-axis if you do not account for that difference. We'll illustrate this fact shortly.

## PLOT

The statement:

PLOT *x_coordinate*, *y_coordinate*

inks in the pixel with these coordinates, so this measly program:

```
10 PLOT INT(RND*128), INT
   (RND*96):INPUT a$: GO TO 10
```

plots a random point each time you press ENTER. This will work on all layers[1], although it will not use the entire area of the screen in all modes. Can you figure out why?

Here is a rather more interesting program. It plots a graph of the function SIN (a sine wave) for values between 0 and 2π:

```
10 FOR n=0 TO 255: REM change
   to 127 for LoRes
20 PLOT n,88+80*SIN(n/128*PI)
30 NEXT n
```

This next program plots a graph of SQR (part of a parabola) between 0 and 4:

```
10 FOR n=0 TO 255
20 PLOT n,80*SQR (n/64)
30 NEXT n
```

Notice that when in *Layer 0*, pixel coordinates are rather different from the line and column in an AT item. You may find the diagrams in *Chapter 14 useful when working out pixel coordinates and line and column numbers for Layer 0*. The other layers as we've already discussed are pretty straightforward. To illustrate, switch to *HiRes* and try again. What you see when entering:

```
 5 LAYER 1,2
10 FOR n=0 TO 255
20 PLOT n,80*SQR (n/64)
30 NEXT n
```

and run the program is exactly what we were talking about earlier. Our graph, has changed both orientation and stops at the middle of the screen's width. To make the out-

1 *All layers, EXCEPT Layer 3 and the High Res versions of Layer 2 as they're not directly supported by NextBASIC.*

put similar to the the first iteration of the program you will need to change the FOR loop and PLOT commands to:

```
10 FOR n= 0 TO 511
20 PLOT n,80*SQR((511-n)/128)
```

This will invert the coordinates to simulate the *Layer 0* display, by drawing inverted, extend the PLOT *x coordinate* to 512 pixels and make sure the PLOT doesn't get out of bounds (that's why we divide by 128 instead of 64). In reality, you do not need to check if you PLOT out of bounds for layers other than *Layer 0*, as graphics commands for these accept locations outside the screen's pixel boundaries, however it's good practice to do so if you want your program to work across layers.

## DRAW and CIRCLE

To help you with your pictures, the computer will draw straight lines, circles and parts of circles for you, using the DRAW and CIRCLE statements.

The statement DRAW to draw a straight line takes the form:

DRAW *x_coordinate*, *y_coordinate*

The starting place of the line is the pixel where the last PLOT, DRAW or CIRCLE statement left off (this is called the PLOT position; RUN, CLEAR, CLS and NEW reset it to the *coordinate 0* of the selected Layer (bottom left hand corner, at (0,0) for *Layer 0*, top left hand corner for all other layers), and the finishing place is x pixels to the *RIGHT* of that and y pixels *UP* or *DOWN* depending on which layer you're on. This would be *UP* for *Layer 0* and *DOWN* for all other layers. The DRAW statement on its own determines the length and direction of the line, but not its starting point.

Experiment with a few PLOT and DRAW commands, for instance:

```
PLOT 0,100: DRAW 80,-35
PLOT 90,150: DRAW 80,-35
```

Notice that the numbers in a DRAW statement can be negative, although those in a PLOT statement can't. Remember *always,* that the display direction of the DRAW statement changes according to the coordinate system used, ergo which layer you choose is very important. You can also plot and draw in colour, although you have to bear in mind all that were discussed in *Chapter 15*. Depending on the chosen layer, colours may cover the whole of an attribute position instead of individual pixels. Only *LoRes* and *Layer 2* modes offer full individual colour pixel control whereas other layers rely on the attribute used. The following program demonstrates this:

```
10 LAYER 2,0: REM disable Layer
    2
20 FOR m=0 TO 5
30 PROC LayChange (m)
40 BORDER 0:PAPER 0:INK 7:CLS:
    REM black out screen
50 x1,y1=0:REM line start
60 c=1: REM ink, starts with
    blue
70 FOR r = 0 TO 9:REM 10
    repetitions
```

```
 80  x2=INT (RND*256): y2=INT (RND*128):
     REM random line end
 90  DRAW INK c;x2-x1,y2-y1
100  x1,y1=x2,y2: REM next line starts
     where last one finished
110  c+=1:IF c=8 THEN c=1
120  NEXT r
130  PAUSE 0: REM Display inspection
140  NEXT m
150  STOP
1000 DEFPROC LayChange (mode)
1010 IF mode=0 THEN LAYER 0
1020 IF mode=1 THEN LAYER 1,0
1030 IF mode=2 THEN LAYER 1,1
1040 IF mode=3 THEN LAYER 1,2
1050 IF mode=4 THEN LAYER 1,3
1060 IF mode=5 THEN LAYER 2,1
1070 ENDPROC
```

In layers other than *LoRes* and *Layer 2*, you can see how the lines seem to get broader as the program goes on, and this is because a line changes the colours of all the inked-in pixels of all the attribute positions that it passes through. You may also be temporarily perplexed about how the program doesn't crash on *LoRes* given that the selected values can exceed these of the physical resolution (*see line 80*). This would definitely be true for compatibility reasons on *Layer 0*, however on other layers, graphics output off screen is permitted for *x* and *y* values up to *65535*. Note that you can embed PAPER, INK, FLASH (only on layers that this is available or not turned off by enabling the *EnhancedULA* functionality), BRIGHT (idem), INVERSE and OVER items in a PLOT or DRAW statement just as you could with PRINT and INPUT. They go between the keyword and the coordinates, and are terminated by either semicolons or commas.

An extra frill with DRAW is that you can use it to draw parts of circles instead of straight lines, by using an extra number to specify an angle to be turned through; the form is:

DRAW *x_coordinate*, *y_coordinate*, *arc_turn*

*x_coordinate* and *y_coordinate* are used to specify the finishing point of the line just as before and *arc_turn* is the number of radians that it must turn through as it goes; if *arc_turn* is a *positive* it turns to the *left*, while if *arc_turn* is a *negative* it turns to the *right*. Another way of seeing *arc_turn* is as showing the fraction of a complete circle that will be drawn: a complete circle is $2\pi$ radians, so if $a=\pi$ it will draw a semicircle, if $a=0.5*\pi$ a quarter of a circle, and so on.

For instance suppose $a=\pi$ . Then whatever values *x* and *y* take, a semicircle will be drawn. Run:

```
10 PLOT 100,100: DRAW 50,50, PI
```

which will draw this:

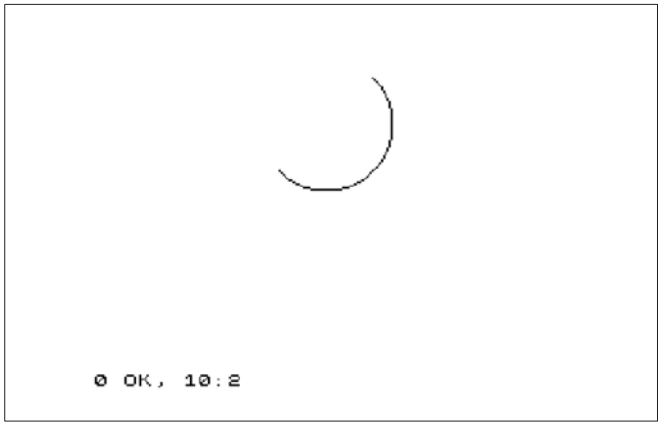



*Fig. 14 – Arc drawn with DRAW statement*

When run on *Layer 0*, the drawing starts off in a south-easterly direction, but by the time it stops it is going north-west: in between it has turned round through **180** degrees, or π radians (the value of **a**). Obviously, when run on other layers, the vertical part of the drawing is inverted in line with everything we have discussed.

Run the program several times, with **PI** replaced by various other expressions e.g. **-PI**, **PI/2**, **3*PI/2**, **PI/4**, **1**,**0**.

**Notes**

Due to the way values are calculated, it's not advisable to use values exceeding π for the *arc_turn* parameter as they may not perform in the way you would intend. That being said there are various values that produce very interesting results. Try:

```
PLOT 75,75: DRAW 80, 24, x
```

where **x** is **400**, **600** or **800**. Experiment further to see what other effects you can generate.

The last statement in this section is the **CIRCLE** statement, which draws an entire circle. You specify the coordinates of the centre and the radius of the circle using:

**CIRCLE** *x_coordinate*, *y_coordinate*, *radius*

Just as with **PLOT** and **DRAW**, you can put the various sorts of colour items in at the beginning of a **CIRCLE** statement. As with its **PLOT** and **DRAW** counterparts, **CIRCLE**, when used in *Layer 0* will produce an error for circles drawn out of bounds but the remaining layers will happily draw off-screen.

## POINT, POINT TO

The **POINT** function informs you of the contents of a pixel on screen. It accepts two parameters enclosed in parentheses, *x_coordinate* and *y_coordinate*. **POINT** on its own works only on *Layer 0* and returns **1** if the pixel is set or **0** if not set. Whilst in *Layer 0* try:

```
CLS: PRINT POINT (0,0): PLOT 0,0
:PRINT POINT (0,0)
```

There's an extended variant of **POINT** utilising the **TO** modifier which works on *all* layers, that takes the output of **POINT** and stores it in variable *var*. This returns **1** if the pixel is set or **0** if not set in all layers except *LoRes* and *Layer 2* just as the plain **POINT** does. In *LoRes* and *Layer 2* however, it returns a value from **0** to **255** which is the actual palette index entry

that the pixel with these coordinates is set to. To illustrate this rewrite the previous example as:

```
CLS: POINT 0,0 TO t: PRINT t: PLOT
0,0:POINT 0,0 TO t: PRINT t
```

Although this may not be the best example for the benefits of using POINT TO instead of the simple POINT, you can save a lot of typing by foregoing a lot of LET statements whilst, at the same time, making your code a lot easier to read *and* working in every graphics mode. It's important to mention that POINT TO *does not* return the contents of a *sprite* that's currently on the given coordinates on screen and instead will return the contents of the layer it's run on.

**Notes**

POINT here is a *function* and not a PRINT *modifier*. Note the distinction as it's important.

## Using OVER and INVERSE with graphics commands

Enter screen mode (EDIT for *NextBASIC Menu* and then the *Screen* option) in the editor and then type:

```
PAPER 7: INK 0
```

and let us investigate how INVERSE and OVER work inside a standard graphics statement. These two affect just the relevant pixel, and not the rest of the character positions. They are normally off (0) in a graphics statement, so you only need to mention them to turn them on (1).

Here is a list of the possibilities for reference:

- PLOT: This is the usual form. It plots an ink dot, i.e. sets the pixel to show the ink colour.
- PLOT INVERSE 1: This plots a dot of ink eradicator, i.e. it sets the pixel to show the paper colour.
- PLOT OVER 1: This changes the pixel over from whatever it was before: so if it was ink colour it becomes paper colour, and vice versa.
- PLOT INVERSE 1; OVER 1: This leaves the pixel exactly as it was before; but note that it also changes the PLOT position, so you might use it simply to do that.

As another example of using the OVER statement fill the screen up with writing using black on white, and then type:

```
PLOT 0,0: DRAW OVER 1;255,175
```

This will draw a fairly decent line, even though it has gaps in it wherever it hits some writing. Now do exactly the same command again. The line will vanish without leaving any traces whatsoever. This is the great advantage of OVER 1. If you had drawn the line using:

```
PLOT 0,0: DRAW 255,175
```

and erased it using:

```
PLOT 0,0: DRAW INVERSE 1;255,175
```

then you would also have erased some of the writing. Now try:

```
PLOT 0,0: DRAW OVER 1;250,175
```

and try to undraw it by:

```
DRAW OVER 1;-250,-175
```

This doesn't quite work, because the pixels the line uses on the way back are not quite the same as the ones that it used on the way down. You must undraw a line in exactly the same direction as you drew it.

Note, that being in *screen mode* in the editor is required for the examples above, otherwise the screen will be reset after each command and you will not get to see the results of the OVER and INVERSE modifiers.

## Using stippling patterns to generate additional colours

One way to get unusual colours is to mix two normal ones together in a single square, using a user-defined graphic. These patterns are called stipples and work reasonably well in lower layers other than *LoRes* (where the pixels are too big) and exceptionally well in *Layer 2* where both the available colours and resolution combine to make the results quite believable. Run this program:

```
1000 FOR n=0 TO 6 STEP 2
1010 POKE USR "a"+n, BIN
     01010101: POKE USR
     "a"+n+1, BIN 10101010
1020 NEXT n
```

which gives the *user-defined graphic* corresponding to a chessboard pattern. If you print this character (*Graphics mode*, then A) in red ink on yellow paper, you will find it gives a reasonably acceptable orange. You can obviously simulate the same behaviour with PLOT statements. This is slower than UDGs but it's much more flexible in the diversity of patterns that you can create.

## Quick erase and fill using LAYER ERASE

*NextBASIC* lacks a dedicated fill command, however large rectangular areas on screen can be filled (or emptied) in *LoRes* and *Layer 2* using the compound LAYER ERASE statement with 4 coordinate parameters (+ 1 optional fill parameter). The command:

LAYER ERASE *x1,y1,x2,y2,c*

will fill the rectangular area delineated by (x1,y1) and (x2,y2) with the *global transparency colour* (if the optional *c* parameter is not specified) or with the colour index contained in the *c* parameter taken from the active palette for the selected layer.

## Clipping windows

One of the nicer features that come as a result of the layer system is the ability to superimpose/combine graphics that exist in separate memory spaces. This is possible on the one hand due to the existence of the transparency colour and on the other hand due to the ability to order the layer superimposition order. The latter is controllable via the LAYER OVER compound command as we saw in the *More on the LAYER command section* in *Chapter 14.*

This can be further enhanced with the creation of clipping windows which are basically smaller areas of a certain layer where all display in this layer goes and leaves the layers underneath visible (without having to set the entire area to be visible to a transparent colour). If you wish to visualise this, imagine a glass window with a rectangular section painted so you cannot see what's behind. That rectangular section is the *clipping window*, in essence the opposite of a regular window. The compound command:

LAYER DIM *x1*,*y1*,*x2*,*y2*

sets the clip window for the current layer from (*x1*,*y1*) to (*x2*,*y2*). Areas of the layer outside this window are not visible. Note that all *Layer 1* modes and *Layer 0* share the same clip window; *Layer 2, Layer 3* and the *Sprite System* have their own separate clip windows. Refer to *Chapter 22* for more information on how clipping windows are defined using the *Next Registers.* The compound command:

LAYER CLEAR

will reset all layer information to defaults. This is also done by NEW. It resets banks, mode, *Layer 2* enable status, layer offsets / clipping windows and layer ordering.

## Tiling

Since straight graphics commands can be slow, *NextBASIC* provides a set of commands that can help recreate parts of, or entire *Layer 2* and *LoRes* screens, very quickly; something that can be very useful especially when a lot of screen elements are being repeated. These screen elements are called *tiles* and much like their real-word counterparts, they are a self-contained graphical rectangular pattern. *Tiles* can be repeated as many times as we need them to, or be completely independent.

Each *tile* can be *8x8 pixels* or *16x16* pixels in size. This allows a 16K *bank* to hold *256* 8x8 tiles or *64* 16x16 tiles. *Tiles* are numbered *0...255*. Therefore, a complete set of 8x8 *tiles* occupies a *single* 16K bank, and a complete set of 16x16 *tiles* occupies *four* 16K banks. If you use 16x16 *tiles*, you can restrict the *tile* number used and therefore reduce the memory requirements (e.g. if you need *64 or fewer* different *tiles*, only *1* 16K bank is required). Additionally for *tiles* to be recalled, a special linear map, called a *tilemap*[2], of 8-bit *tile* numbers is needed. The user can specify any width up to *2048 tiles*; each row of *tiles* follows directly after the previous one.

The *tilemap* must be fully contained inside a single 16K bank. This gives a maximum *tilemap* size of *256x64*, *128x128*, *2048x8* etc.

Any pixels in a *tile* which are the same colour as the current *global transparency colour* will not be written to the screen. If you want to draw pixels containing the *global transparency colour* you can temporarily change it to another colour (not used in your tiles) using the PALETTE OVER command before using TILE. Alternatively, you can use the LAYER ERASE command (see the *Quick erase and fill section* above) to clear regions of the screen to the *global transparency colour* before drawing tiles on top.

*Layer 2* and *LoRes tilemaps* are stored separately, so you can use both simultaneously. The TILE commands affect the currently selected layer/mode. These are:

TILE BANK *n*

which defines bank *n* as containing the *tiles* (up to 4 banks n...n+3 if 16x16 tiles).

TILE DIM *n*,*offset*,*w*,*tilesize*

defines bank *n* as containing the *tilemap*, starting at offset *offset* in the bank. The *tilemap* is width *w* (1–2048) and uses *8x8* (tilesize=8) or *16x16* (tilesize=16) *tiles*.

TILE
TILE AT *x*,*y*

Draws an entire screen from *tilemap*, from *tile offset x,y* in the *tilemap* (0,0 if not specified).

TILE w,h
TILE w,h AT x,y

2 *You may remember that we spoke of tiles before, when initially discussing Layer 3 in Chapter 14. The principle is the same (a repeated rectangular pattern) but the specifics change (9-bit colour vs. 4-bit or 1-bit colour and 16x16 -or- 8x8 pixel tiles vs. ONLY 8x8 pixel tiles).*

TILE w,h TO x2,y2
TILE w,h AT x,y TO x2,y2

The above draw a section of screen from a *tilemap*. Number of *tiles* to draw is width *w*, height *h*. The AT draws from *tile* offset *x*,*y* in the *tilemap* (or 0,0 if not specified as in the previous example), and the TO draws to the *tile* offset *x2*,*y2* on the screen (or 0,0 if not specified).

## Accessing non-supported graphics modes

We spoke previously about how *NextBASIC* does not support the higher resolutions provided by *Layer 2* and the Next hardware. That's not entirely accurate however as there are ways through the power of *Next Registers* (See *Chapter 22*) and BANK POKE (See *Chapter 23*) to do it regardless. There are a few unknown commands in the following short program which you will soon encounter, however the point of the exercise is to show what is possible. For now type it and run it and we'll revisit it in *Chapter 23*:

```
 10 RUN AT 3
 20 REG 112, @10000:REM L2
    SELECT 320x200
 30 REG 24,0:REG 24,159: REG
    24, 0: REG 24,255: REM
    setup clipping window
 40 REG 105,128:REM SHOW L2
 50 FOR %f=0 to 16383
 60 BANK 9 POKE %f, %RND(255)
 70 BANK 10 POKE %f, 30
 80 BANK 11 POKE %f, %RND
    (230)
 90 BANK 12 POKE %f, 60
100 BANK 13 POKE %f, %RND
    (128)
110 NEXT %f
120 PAUSE 0
```

## Exercises

1. Play about with PAPER, INK, FLASH and BRIGHT items in a PLOT statement. These are the parts that affect the whole of the character position containing the pixel. Normally it is as though the PLOT tatement had started off:
   `PLOT PAPER 8; FLASH 8; BRIGHT 8;`
   and only the ink colour of a character position is altered when something is plotted there, but you can change this if you want. Be especially careful when using colours with INVERSE 1, because this sets the pixel to show the paper colour, but changes the ink colour and this might not be what you expect.
2. Try:
   `CIRCLE 100,87,80: DRAW 50,50`
   You can see from this that the CIRCLE statement leaves the PLOT position at a rather indeterminate place – it is always somewhere about halfway up the right hand side of the circle. You will usually need to follow the CIRCLE statement with a PLOT statement before you do any more drawing.

# Chapter 17 – Time and Motion

One of the most important features of the ZX Spectrum Next is the ability to move things on screen fast, either via the usage of sprites or by quickly interchanging full screens to create animations and general visual effects. Motion (and animation) however, as on real life, is a function of time. In other words we need to precisely count time in order to display things and for this purpose this chapter will deal with these two seemingly unrelated subjects in one unit. We will begin with the whole idea of timekeeping on the computer and all the facilities the ZX Spectrum Next has in order for us to measure time.

Timekeeping is essential in computing as all devices work on the basis of a unit of time (in our case Hertz –or– Hz) but much of this happens behind the scenes. Here we will examine commands related to time together with the optional timing hardware, before we move into animation, scrolling, the Sprite Engine and eventually to the Copper.

## PAUSE

While the general attitude in programming is to make things execute as fast as possible, we often find ourselves in need of making our program wait for a specific length of time or even indefinitely. There is a number of reasons why that would be the case; expecting user interaction is one; displaying warnings is another, timing precisely something is a third and for all the above and more you will find the **PAUSE** statement useful.

### PAUSE *n*

stops computing and displays the picture for ***n*** frames of the selected display mode.

In 50Hz mode, there are 50 *frames-per-second* (*fps*), so setting *n* to **50** would result in **1** sec. pause. Respectively in 60Hz mode which runs at 60 *fps* this figure would be **60** for **1** sec. pause.

These modes are set these at the Configuration boot menu or via the **config.ini** file which is located in the **c:/machines/next/** folder. Generally speaking, almost all modern HDMI™ and VGA displays operate at 60Hz, while many also have 50Hz modes.

*n* can be up to **65535**, which gives you just a little over **21** minutes at 50Hz and just under **19** minutes at 60Hz respectively; if n is set to **0** then it means **PAUSE** indefinitely.

A pause of any length (including the indefinite ones) can always be cut short by pressing a key (note that **CAPS SHIFT** + **Space** or **BREAK** will cause a break as well). You have to press the key down after the pause has started.

This program works the second hand of a clock:

```
 10 REM We select the appropriate
    pause
 20 wait=52:REM 50Hz/50=1 sec.
 30 REM First we draw the clock face
 40 FOR n=1 TO 12
 50 PRINT AT 10-10*COS(n/6*PI),
    16+10*SIN(n/6*PI);n
 60 NEXT n
 70 REM Now we start the clock
 80 FOR t=0 TO 200000: REM t is the
    time in seconds
 90 a=t/30*PI : REM a is the angle of
    the second hand in rad.
100 sx,sy=80*SIN a,80*COS a
200 PLOT 128,88: DRAW OVER 1;
    sx,sy: REM draw 2nd hand
```

```
210 PAUSE wait
220 PLOT 128,88: DRAW OVER 1;
    sx,sy: REM erase 2nd hand
400 NEXT t
```

This clock will run down after about 55.5 hours because of line 60, but you can easily make it run longer. Note how the timing is controlled by line 20. When running in *50Hz* mode, you might expect PAUSE 50 to make it tick one a second, but the computing takes a bit of time as well and has to be allowed for. This is best done by trial and error, timing the computer clock against a real one, and adjusting line 20 until they agree. (You can't do this very accurately; an adjustment of one frame in one second is 1.67% or less than half an hour in a day.)

## Using POKE and PEEK at the System Variables

There is a much more accurate way of measuring time. This uses the contents of certain memory locations. The data stored is retrieved by using PEEK. *Chapter 24 – The System Variables*, explains what we're looking at in detail. The expression used is:

```
(65536*PEEK 23674+256*PEEK 23673+PEEK 23672)/50
```

which gives the number of seconds since the computer was turned on (up to about 3 days and 21 hours, when it goes back to 0). That being said *System Variables* are not guaranteed to be in the same location or even accessible with succesive versions of *NextBASIC* and *NextZXOS*. For that reason, we are provided with one hybrid command/function which does the exact same thing removing uncessecary calculations and non-standard access to the system variables. The keyword in question is TIME and we'll examine it below.

## TIME (command/function)

When used as a command, TIME takes no arguments and simply resets the frame counter (*System Variable* FRAMES – See *Chapter 24* for details) to 0. Intended for use with the corresponding TIME function which returns how many frames have passed since bootup or since the FRAMES counter was reset. For example here's a silly program to verify PAUSE and TIME measure the same thing:

```
10 INPUT "How many PAUSE
   frames? ";f
20 TIME
30 PROC perftest(f)
40 PRINT '"perftest() took
   ";TIME;" frames against
   desired ";f;" frames."
50 STOP
60 DEFPROC perftest(frames)
70 PAUSE frames
80 ENDPROC
```

## Retrieving information from the RTC

Your ZX Spectrum Next has a DS1307 *Real Time Clock* (*RTC*) installed, which allows you to use a more accurate way of retrieving timekeeping data; one that doesn't involve any calculations as described above; nor one that can be affected by clock speed changes.

There are two ways to retrieve time (or date) information from the *RTC*. The first is not very straightforward owning to the fact that it's triggered via a *dot command*. The second how-

ever is via *NextBASIC* and it's the aptly named function TIME$. Let's examine both as the first method can be used for several other *NextZXOS* facilities that return information for which *NextBASIC* doesn't yet have a specialised keyword.

We need to use the *NextZXOS* facilities of *Channels* and *Streams* (which we will explore in *Chapter 20*) and specifically, *Channel* v (which opens a *stream* to a fixed sized variable t$)

```
DIM t$(100):OPEN #2,"v>t$":.TIME
:CLOSE #2:PRINT t$
```

then by string slicing t$ as seen in depth in *Chapter 7*, we can extract the information we need to use .time (or .date) in our programs.

### TIME$

Function TIME$ does the exact thing as the example above but in a much cleaner way and moreover, the user does not need to get only time or date separately as with TIME$ you get them both simultaneously.

For example typing:

```
PRINT TIME$
```

will return a string of the format *yyyy-mm-dd hh:mm:ss* like:

```
2023-05-10 16:55:32
```

which is obviously easy to slice and get the information from.

Now we already discussed that a frame lasts a different amount of time depending on if your computer is running at 50Hz or at 60Hz. Here's a revised program that finds out what frequency you're running on first and then tests against the amount of PAUSE frames you'll request. Do not pay attention to the REG function, we will explain it in *Chapter 22*.

```
 10 sp=REG 5&@100>>2
 20 PRINT "I'm running at
    ";sp?("50","60");Hz"
 30 INPUT "How many PAUSE
    frames? ";f
 40 PRINT
 50 PROC gettime() TO H1,M1,S1
 60 PRINT "Testing at
    ";sp?("50","60"); "Hz
    started
    at:";H1;":";M1;":";S1
 70 TIME
 80 PROC PerfTest(f)
 90 PRINT '"PerfTest() took
    ";TIME;" frames against
    desired ";f;" PAUSE
    frames."
100 PROC GetTime() TO H2, M2,
    S2
```

```
110 PRINT "Testing at
    ";sp?("50","60"); "Hz
    ended
    at:";H2;":";M2;":";S2
120 STOP
130 DEFPROC perftest(frames)
140 PAUSE frames
150 ENDPROC
160 DEFPROC GetTime()
170 a$=TIME$
180 h,m,s=VAL(a$ (12 TO 13)),
    VAL(a$(15 TO
    16)),VAL(a$(18 TO))
190 ENDPROC = h,m,s
```

Line 10 reads the *Next Register bit* that holds the vertical frequency to see if it's 50 or 60Hz before asking you how many PAUSE frames you want to test against. The next thing that happens is that procedure GetTime is called which gets the current time in a string formatted as described above in the section devoted to TIME$. By using string slicing and the VAL function we can get the hour, minute and second separately in case we need to manipulate or use them later. Immediately afterwards, the TIME command is initiated which as we mentioned above resets the frame counter. Following that, we call procedure PerfTest which performs a PAUSE for as many frames as we've asked already for and then we inform the user if the requested PAUSE frames and the actual frames as returned by function TIME match. Finally we get once again the current time, store it in variables h2,m2 and s2 and display it. If you run the program for enough frames and you perform some arithmetic using s1, m1, h1 and s2, m2 and h2 you'll easily figure out how much time a frame as requested by PAUSE lasts.

Here is a revised clock program to make use of this:

```
 10 REM First we draw the clock face
 20 FOR n=1 TO 12
 30 PRINT AT 10-10*COS(n/6*PI),
    16+10*SIN(n/6*PI);n
 40 NEXT n
 50 DEF FN secs()=VAL TIME$(18 TO):
    REM Get the number of seconds
100 REM Now we start the clock
110 t1=FN secs()
120 a=t1/30*PI: REM a is the angle of
    the second hand in radians
130 sx,sy=72*SIN a,72*COS a
140 PLOT 131,91: DRAW OVER 1;sx,sy:
    REM draw hand
200 t=FN secs()
210 IF t<=t1 AND t<>0 THEN GO TO 200:
    ELSE IF t=0 and t1 >t then go to
    220: else go to 220: REM wait
    until time for next hand except
    if seconds reset to 0
```

```
220 PLOT 131,91: DRAW OVER 1;sx,sy:
    REM rub out old hand
230 t1=t: GO TO 120
```

The real time clock that this method uses should be accurate to about .001% regardless if the computer is just running its program, or about 1 second per day; unlike the PEEK method shown above where the computer (and the counter) stops temporarily whenever you do BEEP, or a storage device operation, or use the printer or any of the other extra pieces of equipment you can use with the computer. All these would make it lose time however using the RTC which runs independently bypasses this issue.

If you have chosen to run your ZX Spectrum Next in 60Hz mode then for any program that uses PAUSE you must replace 50 by 60 where appropriate.

### INKEY$

The function INKEY$ (which has no argument) reads the keyboard. If you are pressing exactly one key (or a SHIFT key and just one other key) then the result is the character that that key gives in L mode; otherwise the result is the empty string.

Try this program, which works like a typewriter:

```
10 IF INKEY$ <>"" THEN GO TO 10
20 IF INKEY$ = "" THEN GO TO 20
30 PRINT INKEY$;
40 GO TO 10
```

Here line 10 waits for you to lift your finger off the keyboard and line 20 waits for you to press a new key.

Remember that unlike INPUT, INKEY$ doesn't wait for you. So you don't type ENTER, but on the other hand if you don't type anything at all then you've missed your chance.

INKEY$ is very useful for a control loop where you can set objects on the screen to move according to which key you're pressing (for example the *cursor* keys). As you will also see from *Chapter 20*, one more option for you is to use the NEXT #...TO keyword that works in a very similar manner. Finally it's also possible to query the keyboard hardware directly as well as the optional mouse as you will see in *Chapter 22*.

## Animation: a quick primer

Animation is defined as any process with which static objects or pictures are manipulated to appear as moving. The word itself comes from the Latin *anima* which means *life*. In essence, it is to convey the appearance of life and movement to otherwise static constructs.

In computers, this is achievable using the rapid succession of images faster than the eye can perceive. On the ZX Spectrum Next specifically, there are basically five methods of animation; one using *mass storage frame playback*, the other using *memory based frame playback*, the third using *sprites*, the fourth using *scrolling* and the fifth is to use a combination of all the above. Let's examine them in turn.

### Mass Storage Frame Playback

This technique deals with restoring partial or complete frames of screens stored on your SD card to or RAMdisk to the screen memory in rapid succession at the maximum possible speed. Consider this example using the RAMdisk:

```
10 INK 5: PAPER 0: BORDER 0: CLS
20 FOR f=1 TO 10
30 CIRCLE f*20,150,f
40 SAVE "m:ball"+ STR$ (f) CODE
   16384,2048
```

```
 50 CLS
 60 NEXT f
 70 FOR f=1 TO 10
 80 LOAD "m:ball"+ STR$ (f) CODE
 90 NEXT f
100 BEEP 0.01, 0.01
110 FOR f=9 TO 2 STEP -1
120 LOAD "m:ball"+ STR$ (f) CODE
130 NEXT f
140 BEEP 0.01, 0.01
150 GO TO 70
```

The example above works only on Layer 0 and leverages the RAMdisk without getting into BANK management territory. It can do that because the frames we're saving are very small. If you remember from Chapters 14 through 16 how the Layer 0 memory is organised in thirds, you'll soon figure out that although small it's not necessarily the faster way of doing things.

The RAMdisk is good to replay things but our SD card is also quite good. Let's try the following example with something more complicated based on a program contributed by mathematician Uwe Geiken from the *NextBASIC* forum.

```
  1 REM Based on Rotating Ellipses by
    Uwe Geiken © 2019
 10 RUN AT 3
 20 LAYER 2,1: PAPER 0: CLS
 30 X,Y=128,88
 40 A,B=20,0
 50 ITER,CURITER=20,0
 60 FOR Q=0 TO 2* PI STEP PI/ITER
 70 INK 246: A,B=30,16: P=Q: PROC
    ellipse (X,Y,A,B,P)
 80 INK 155: A,B,P=19,10,2* PI-Q:
    PROC ellipse (X,Y,A,B,P)
 90 IF CURITER <=ITER THEN SAVE
    "ANIM"+STR$ (CURITER)+",SL2"
    LAYER: CURITER +=1
110 PRINT AT 23,0; "Frame:";
    CURITER-1;" saved";:CLS: IF
    CURITER > ITER THEN GO TO 220
120 NEXT Q: GO TO 220
130 DEFPROC ellipse (X,Y,A,B,P)
140 LOCAL c,d,i,j,k,s
150 c,d=COS P,SIN P
160 FOR k= 0 TO 2.05* PI STEP PI /20
170 i,j=A* COS k,B* SIN k
180 IF k=0 THEN PLOT x+i*c-j*d,
    y+i*d+j*c: GO TO 200
190 DRAW x+i+*c-j*d- PEEK 23428,
    y+i*d+j*c- PEEK 23430
200 NEXT k
210 ENDPROC
220 FOR %I = 0 TO 5
```

```
230 FOR J= 0 TO ITER
240 LOAD "ANIM"+ STR$ (J)+".SL2"
    LAYER
250 NEXT J
260 NEXT %I
360 LAYER 2,0: LAYER 0
```

The program generates ellipses that rotate counter to one another and after drawing each frame, saves the entire screen on the SD card. Once it's done generating (when CURITER reaches ITER), it uses LOAD … LAYER (which we will look at in depth in *Chapter 19*) to load and display the *Layer 2* screens the previous part generated. Unlike the previous example using *Layer 0* which only moved 2K at a time, this loads and displays 48K at a time.

Compared to the previous example using the *RAMdisk*, this appears much smoother and the reason is simple; there are many more frames generated by the program than what the previous one did. The question is can it be made smoother and if at all possible, faster?

## Memory Based Frame Playback

It's time to delegate frame playback to RAM. Replace line 90 with this, longer, version:

```
 90 IF CURITER <=ITER THEN SAVE
    "ANIM"+STR$ (CURITER)+",SL2"
    LAYER: BANK 9 COPY TO
    111-(CURITER*3): BANK 10 COPY TO
    110-(CURITER*3): BANK 11 COPY TO
    109-(CURITER*3): CURITER +=1
```

and then add the following lines at the end:

```
270 PRINT AT 22,0;"Done Loading
    from SD. Press any key to
    load from memory"
280 PAUSE 0
290 FOR %I=0 TO 5
300 FOR %J=0 TO % INT (ITER)
310 BANK %111-(3*J) COPY TO %9
320 BANK %110-(3*J) COPY TO %10
330 BANK %109-(3*J) COPY TO %11
340 NEXT %J
350 NEXT %I
360 LAYER 2,0: LAYER 0
```

Run the program again and now compare the playback using the SD card, with the playback of all the screens using the memory.

You can see that the playback is even smoother AND faster than the SD card and the reason is simple and that is because memory is a much faster medium than your SD card. Now there are several things of note here. First of all, this is not very efficient code, memory wise; *Layer 2* uses 3 banks of 16K each making an entire screen 48K long. For the 20 iterations we made, that's 20 * 3 * 16K = 960K making this program unlikely to work on a non-expanded KS1 ZX Spectrum Next[1]. Secondly, not the entire screen is moving. Only a small window does and that makes saving the remainder of each screen wasteful in mem-

1 *If you modify variable ITER however to a value around 10 it will work since we already know that banks 0 to 12 are being used by the system and 10*3*16 gives us a figure of 480K which is a memory size available on an unexpanded Next.*

ory and speed. If we modify the program to confine the ellipses in one third of the screen (vertically speaking), we we can only use 16K at a time making the program playback much faster. This is essentially the same thing the first program did using the *RAMdisk*. That one however appears jerky because there are not enough frames of animation to make our eyes be fooled by the illusion of smooth movement.

We can do that using BANK LAYER which is used to quickly copy data from a memory bank to the screen or vice versa*. The syntax is as follows:*

BANK *n* LAYER *x,y,w,h*|*offset* TO [*raster_op*] *offset*|*x,y,w,h*

which can copy any rectangular "window" of the current *layer* defined by *x*,*y*,*w* and *h* into a memory bank and back. BANK LAYER also supports effects defined by *raster_op* which can further enhance the display of the "window" you're copying making animation transitions even more interesting. More information regarding BANK … LAYER can be found in *Chapter 23 – The Memory*.

## Animation with the Sprite System

The third way of animating things in *NextBASIC* is via the use of the *Sprite System*. *Sprites* are visual objects of a rectangular shape that can be placed anywhere in the screen and animated by moving them about but also perform animation within the object by rapidly replacing the object's bitmap (the image –or *pattern*– it displays). There are two kinds of *sprites* on the ZX Spectrum Next, 8-bit and 4-bit. The first can display 256 colours at once while the second 16.

There is a maximum of 128 *sprites* and 64 *sprite patterns* in 8-bit mode and 128 in 4-bit mode. *NextBASIC* only supports the 8-bit mode *sprites* so we'll only discuss these. For more information regarding the use of 4-bit *sprites*, refer to *Chapter 22* and online at specnext.com. Information on 4-bit sprites is also included in the second volume of this manual.

*Sprites* are 16 x 16 pixels in size and can be mirrored and rotated. They can also be anchored together to make a bigger sprite.

The *Sprite System* has it's own RAM, located inside the FPGA that's at the core of the computer, which not accessible from the outside via standard PEEK and POKE; one can only write to it via REG commands and the special *sprite ports* (See *Chapter 22 for details), so we need to keep a copy of our sprites* in memory if we want to modify and send them to be displayed anew.

## Creating Sprites

Sprites are created very similar to the way UDGs are created as we saw in Chapter 13.

There are three major differeces however:

- UDGs are 1-bit only while sprites (for *NextBASIC*) are 8-bit
- UDGs are 8 x 8 while sprites are 16 x 16 pixels
- UDGs are manipulated within the main memory map while sprites need to be stored in a bank in order to be used.

The similarities however are obvious. Sprites can be easily made with DATA statements which –if using one of the wider display modes– can even be seen visually via the numbers.

So where for a UDG you wrote 8 DATA statements of 8 bits each, for a sprite you write 16 DATA statements of 16 bytes each; the same essential thing but scaled up.

This is best demonstrated visually so, let's try to implement the following sprite via DATA statements:



*Fig. 15 – A sprite*

The transparency (the large magenta-coloured area) is set to index 227 (as we've seen in *Chapter 15), the Global Transparency Colour* – which for the purposes of our example has been left the default. The rest displays a little spaceship in brown and grey while the cockpit is demonstrated in blue and white.

Let's start with the DATA statements. Some line numbers are omitted as we'll be adding them in the course of our animation example

```
 10 ; Sprite: Romylos Dokos © 2019
 30 RESTORE
 40 BANK NEW a
 50 FOR F=0 TO 255
 60 READ n: BANK a POKE f,n
 70 NEXT f
 80 SAVE "spaceship.spr" BANK a,0,256
210 REM Sprite Pattern 0
220 DATA 68, 68, 68, 68, 227, 227,
    227, 227, 227, 227, 227, 227, 68,
    68, 68, 68
230 DATA 68, 182, 219, 68, 227, 227,
    227, 68, 68, 227, 227, 227, 68,
    219, 182, 68
240 DATA 68, 68, 68, 68, 227, 227,
    227, 55, 55, 227, 227, 227, 68,
    68, 68, 68
250  DATA 182, 182, 68, 227, 227,
    227, 227, 55, 55, 227, 227, 227,
    227, 68, 182, 182
260 DATA 68, 68,  68, 227, 227, 227,
    68, 68, 68, 68, 227, 227, 227,
    68, 68, 68
270 DATA 240, 68, 68, 227, 227, 227,
    68, 255, 127, 68, 227, 227, 227,
    68, 68, 240
280 DATA 227, 68, 68, 0, 227, 227,
    68, 127, 127, 68, 227, 227, 0,
    68, 68, 227
290 DATA 227, 182, 219, 72, 0, 227,
    182, 0, 68, 68, 227, 0, 72, 219,
    182, 227
```

```
300 DATA 227, 182, 219, 72, 182, 0,
    0, 0, 68, 182, 227, 182, 72, 219,
    182, 227
310 DATA 227, 182, 219, 72, 182, 68,
    68, 0, 68, 68, 68, 182, 72, 219,
    182, 227
320 DATA 227, 240, 68, 72, 182, 68,
    68, 0, 68, 68, 68, 182, 72, 68,
    240, 227
330 DATA 227, 227, 227, 72, 182, 68,
    255, 182, 182, 255, 68, 182, 72,
    227, 227, 227
340 DATA 227, 227, 227, 227, 68, 68,
    255, 68, 68, 255, 68, 68, 227,
    227, 227, 227
350 DATA 227, 227, 227, 227, 227, 68,
    255, 182, 182, 255, 68, 227, 227,
    227, 227, 227
360 DATA 227, 227, 227, 227, 227,
    227, 236, 224, 236, 224, 227,
    227, 227, 227, 227, 227
370 DATA 227, 227, 227, 227, 227,
    227, 227, 252, 252, 227, 227,
    227, 227, 227, 227, 227
```

If you use the 64 or 85 column modes (Via the *Edit/Options menu*) you'll be able to discern the pattern in a similar manner as you did for the UDGs in *Chapter 13. Value* 227 is obviously the transparency as we discussed above.

Line 40 is a new command for us (which we will examine in length in *Chapter 23)* but what it does, is to reserve the first free *memory bank* and assign its identification number to variable *a*. This way we don't need to remember –or hard code– an arbitrary number as that number could be in use if this is loaded on another machine.

Next, line 60 reads each value in succession and then writes (with BANK POKE) each value in a progressively increasing *offset* in bank a. Once the READ process is done, we SAVE the stored values in a file for later use. This particular version of SAVE (SAVE ... BANK) will be explained in length in chapters *19 and 23.*

## Putting Sprites on Screen

The sprite (or rather a *pattern* that can be assigned to a sprite) is now safely stored in bank a. So how do we display it?

For that we need a few commands. SPRITE CLEAR, SPRITE BANK, SPRITE PRINT, SPRITE BORDER and finally SPRITE.

Let's follow them one by one:

### SPRITE CLEAR

clears all sprite assignments and starts fresh. It's a good idea to start any program dealing with sprites with that command so let's insert it into our program immediately with:

```
20 SPRITE CLEAR
```

We now have let *NextBASIC* know that we have no sprites assigned with the previous command, but now we need to assign new ones. This is done with:

SPRITE BANK *b [,o, p, n]*

which lets *NextBASIC* know in which bank *b*, are the sprite *patterns* located. Optionally you can define a number *n* of sprite *patterns* beginning with *pattern p*, located at bank *offset o*.

In the case above, we already know the bank and we do not need any more identification factors so let's tell *NextBASIC* where we put the sprites by adding:

```
90 SPRITE BANK a
```

All is now left to do, is show our sprite. For this we need two commands. First we need to enable sprites with:

SPRITE PRINT *n*

where *n* can be **0** or **1** enables sprites (**1**) or disables (**0**) them. This is actually showing the sprites, but freshly initialised sprites contain no image (*pattern*), nor display information. We need to assign at least one *pattern* to one sprite "slot" and tell the Sprite System that the particular sprite "slot" is visible for that to happen.

In our example so far (that will soon change), we only have one pattern so that's not particularly difficult. We also need to place the sprite somewhere on the screen AND possibly rotate it. If you go back to our sprite design, you'll see it's a spaceship facing upwards; we may need to make it turn to the left or right. All of the above (and one more thing) can be achieved with a single command:

SPRITE *s, x, y , p, f,rf,mx,my*

which in one go: sets sprite number *s*, to pattern number *p*, then update its position to location *x*, y with flags *f,* relative flags *rf,* x-scaling *mx* and y-scaling *my.*

If the sprite id s is negative, this sprite is a relative sprite, and its position is relative to the previous anchor sprite (defined with a positive sprite id). See later for more information on relative sprites. Flags is a bitmask (we've covered bitmasks before in *Chapter 6* so that should be easy already) that sets the following:

Bit **0** is the *visibility* flag. **0** is for invisible and **1** is for visible
Bit **1** is the *rotate* flag. **0** for standard, **1** for a 90° clockwise rotation
Bit **2** is the *Y–mirror* flag. **0** is for non-mirrored vertically while **1** is for mirrored
Bit **3** is the *X-mirror* flag. Again it's **0** for non-mirrored horizontally while **1** is for mirrored

while

Bits **4** through **7** define a 4-bit *palette offset* (or **0**).

The Relative Flags *rf* is also a bitmask that sets the following:

The relative-flags parameter, rf, is also a bitmask:

Bit **0**: type: **0**=composite, **1**=unified [only valid for anchor sprites]
Bit **1**: pattern is relative to the anchor [only valid for relative sprites]
Bit **2**: palette offset is relative to the anchor [only valid for relative sprites]

The scaling parameters *mx* and *m*y are:

0=1x (no scaling)
1=2x
2=4x
3=8x

Any parameters in the SPRITE command can be omitted, and its value will be left unchanged from the last time it was explicitly specified. It's a good idea again to use the BIN function to easily specify the flag parameters in a more convenient way.

We'll explain in a little bit the part about the *palette offset* but for now, let's add a non-mirrored, non-rotated sprite **0** with the pattern **0** we defined, put it at approximately the centre

of our screen and make it visible. Let's add the appropriate commands now to our program:

```
100 SPRITE PRINT 1
130 SPRITE 0,152,119,0,1
```

to make sure that our sprite will stay on screen (as the *NextBASIC* editor will make it invisible temporarily when invoked), we should add one more line:

```
150 PAUSE 0
```

which will ensure the computer is waiting on our keypress before returning to *NextBASIC*. Now RUN the program.

Presto! Our Spaceship is sitting idle, doing nothing, in the middle of our screen. But wait a second? 152 and 119 don't look anywhere like the middle of the screen. We know our resolution in *Layer 0* can be expressed in values between 0 and 255 for *x* and 0 and 191 for *y* correct? Well wrong! It's time now to refer back to *Chapter 15* and also examine *Fig. 13* one more time where we will see that the Sprite System has a resolution of 320 w x 256 h pixels.

This gives us 32 more pixels on every side than our standard resolution *Layer 0* and *Layer 2* screens. Now placement of the sprite begins with the upper left corner and a sprite is 16 x 16 so in order to be placed at the centre of the screen you divide the horizontal and vertical in half and then subtract a further 8 pixels to center the sprite. Normally the border hides the sprites so setting an *x,y* set of 0,0 would leave the sprite invisible. There is something we can do about that however and that's use:

SPRITE BORDER *n*

which sets the sprites to print over the border if *n* is set to 1 or under it if *n* is set to 0. Let's try it by adding the command and changing line 140 to show the sprite at that coordinate with:

```
105 SPRITE BORDER 1
130 SPRITE 0,0,0,0,1
```

To execute with the latest changes, do not RUN the program again, as this will repeat the process and commit one more bank to the sprite DATA we entered originally. Instead type GO TO 100. You may even want to test this without line 105 to see the difference. One more command related to the above is:

SPRITE DIM *x1,y1,x2,y2*

which sets the clip window for sprites from (*x1,y1*) to (*x2,y2*). Any part of a sprite outside this window is not visible. Note that this has no effect if sprites over the border (SPRITE BORDER 1) is enabled.

## Animating Sprites

This chapter however is called Time and Motion and with sprites so far we haven't seen motion at all! Well, let's change that; as we spoke in the introduction a sprite can be animated by moving it about the screen or by changing its bitmap to something different and most of the time, both at the same time. In order however to animate the bitmap of a sprite, a new pattern has to be defined. Let's do that by adding a few lines to our program and modifying some existing ones. First remove lines 140 and 150, then modify these:

```
 50 FOR F=0 TO 511
 80 SAVE "spaceship.spr" BANK a,0,512
```

and then add these:

```
106 FOR %a= 1 TO 50
139 %s=1-s
```

```
140 SPRITE 0,152,119,%s,1
145 NEXT %a
150 PAUSE 0:STOP:REM Exit here after
    pausing
380 REM Sprite Pattern 1
390 DATA 68, 68, 68, 68, 227, 227,
    227, 227, 227, 227, 227, 227, 68,
    68, 68, 68
400 DATA 68, 219, 182, 68, 227, 227,
    227, 68, 68, 227, 227, 227, 68,
    182, 219, 68
410 DATA 68, 68, 68, 68, 227, 227,
    227, 55, 55, 227, 227, 227, 68,
    68, 68, 68
420 DATA 182, 182, 68, 227, 227, 227,
    227, 55, 55, 227, 227, 227, 227,
    68, 182, 182
430 DATA 68, 68, 68, 227, 227, 227,
    68, 68, 68, 68, 227, 227, 227,
    68, 68, 68
440 DATA 240, 68, 68, 227, 227, 227,
    68, 255, 127, 68, 227, 227, 227,
    68, 68, 240
450 DATA 227, 68, 68, 0, 227, 227,
    68, 127, 127, 68, 227, 227, 0,
    68, 68, 227
460 DATA 227, 182, 219, 72, 0, 227,
    182, 0, 68, 68, 227, 0, 72, 219,
    182, 227
470 DATA 227, 182, 219, 72, 182, 0,
    0, 0, 68, 182, 227, 182, 72, 219,
    182, 227
480 DATA 227, 182, 219, 72, 182, 68,
    68, 0, 68, 68, 68, 182, 72, 219,
    182, 227
490 DATA 227, 240, 68, 72, 182, 68,
    68, 0, 68, 68, 68, 182, 72, 68,
    240, 227
500 DATA 227, 227, 227, 72, 182, 68,
    255, 182, 182, 255, 68, 182, 72,
    227, 227, 227
510 DATA 227, 227, 227, 227, 68, 68,
    255, 68, 68, 255, 68, 68, 227,
    227, 227, 227
520 DATA 227, 227, 227, 227, 227, 68,
    255, 182, 182, 255, 68, 227, 227,
    227, 227, 227
530 DATA 227, 227, 227, 227, 227,
    227, 236, 224, 236, 224, 227,
    227, 227, 227, 227, 227
```

```
540 DATA 227, 227, 227, 227, 227,
    227, 227, 224, 224, 227, 227,
    227, 227, 227, 227, 227
```

Now, unlike the previous encouragement, **RUN** the program again. This will reserve a new bank for sprites which isn't normally recommended but it is okay for the purposes of our example. What we have done now is to create two patterns that are similar but differ slightly in the cannons section and the engine section. Lines 136 to 150 will display sprite **0**, 50 succesive times, however where things differ is at line 137 which "flips a switch" from pattern **0** to pattern **1** for sprite **0** displayed at line 140. If you cannot see the effect very well, you can insert a:

```
PAUSE 3
```

at the end of line 140 which should give you just about enough delay to see the sprite changing at the engine and cannon sections while at the same time demonstrating how important time control is in animation. We did cover the bitmap animation of the sprite itself; let's now see how we can make it move. First however let's try to rotate the sprite in place so we can also see the usage of the flags in action. Add the following lines:

```
107 %p=0
108 REPEAT
109 IF %p=0 THEN %f=%00001
110 IF %p=1 THEN %f=%00011
111 IF %p=2 THEN %f=%00101
112 IF %p=3 THEN %f=%01011
141 REPEAT UNTIL %p>3
```

and make line 140:

```
140 SPRITE 0,152,119,%s,%f:PAUSE 3:
    %p+=1
```

Now execute again with **GO TO 100** and you will see the sprite rotate in place.

**Notes**

SPRITES cannot be saved as parts of any screenshot facility with the *NMI menu* or via SAVE … LAYER because they exists outside of normal memory space.

The process is quite simple; the last bit being the visibility flag:

First the sprite is printed upright, then the *rotation flag bit* gets turned on to give it a right angle turn, then it gets turned off and the *Y mirror flag bit* gets turned on to make the sprite point downwards and finally the *rotation flag bit* together with the *X mirror flag bit* get turn on to rotate the sprite clockwise 90° and then mirrored horizontally to make the sprite pointing to the left. The process restarts from the sprite pointing upwards when the rotation variable **%p** gets reset to **0** and the whole thing repeats **50** times, all the while changing between patterns **0** and **1**.

## Moving Sprites on Screen

Time to move the sprite about the screen; we'll start easy and then introduce you to the real reason (that is obviously humourus) why maths exist! First remove all lines between 106 and 150 and replace with these:

```
106 FOR %a = 0 TO 255
```

```
130 %s=%1-s
140 SPRITE 0,152, %255-a,%s,1
141 PAUSE 3
145 NEXT %a
150 GO TO 106: REM you'll need to
    stop this with BREAK
```

Execute with GO TO 100 and you'll see our spaceship fire up its engines and cross the screen from top to bottom. Now for something much fancier as promised, move line 106 to 120 and add these lines:

```
106 PROC initSXSineMov()
560 STOP
570 DEFPROC initXSineMov()
580 FOR f=0 TO 319:%a[ INT {f}]=% INT
    {159* SIN (f/159* PI )}: NEXT f
590 ENDPROC
```

Finally modify lines 140 and 150 as follows:

```
140 SPRITE 0,%159+a[a], %255-a, %s,1
150 GO TO 120
```

before executing again with GO TO 100. The spaceship now will move in a sinusoidal pattern from the bottom to the top of the screen before wrapping around and coming from the bottom. The way we did this, was by precalculating an integer array (See *Chapter 12*) to hold all possible x values within our visible Sprite System coordinates. To avoid B Integer out of range errors, we made sure the possible values of both the SIN function results and line 140 that positions the spaceship in the *x,y* axis stay within acceptable range. To switch the initial direction of movement, instead of a + you can start with a - in line 140 as follows:

```
140 SPRITE 0,%159-a[a],%255-a,%s,1
```

Note that our integer array %a is using the brackets [ ] variant instead of the parentheses ( ) variant and that's because we have more than a potential 64 values. That means also that integer arrays %a ( ), %b ( ), %c ( ), %d ( ) and %e ( ) have been used up by %a[ ].

It's obvious by this example that very complex animation patterns can be created with relative ease using the Sprite System. Before we move on to scrolling, it's useful to also cover a few more subjects we did not address in the course of our example.

The first thing is the ability to use palettes with the Sprite System. These are indistinguishable from other palettes in the ZX Spectrum Next palette control system[2] and they too are also governed by the PALETTE DIM keyword to set them up as 8 or 9 bit. Like the LAYER PALETTE equivalent, the Sprite System has its own keyword combinations: SPRITE PALETTE and SPRITE PALETTE BANK. Their syntax is as follows:

SPRITE PALETTE *n[,i,v]*

where *n* is the palette number (0 for first and 1 for second) while the optional *i*, *v* are the colour index (0 to 255) and colour value (expressed in 9-bit RRRGGGBBB format regardless of the PALETTE DIM setting).

SPRITE PALETTE *n* BANK *b, o*

will operate like it's LAYER counterpart, assigning palette *n* from offset *o* in bank *b*. As with the LAYER version, palettes are 512 bytes long if 9-bit and 256 bytes long if 8-bit (as set with PALETTE DIM).

2 *See Chapter 15 for the Layer 2 notable palette exception*

One last thing of note is the palette offset flag we discussed earlier. This is there to allow for quick change of colour scheme on a sprite without changing its bitmap. If you recall the discussion about 4-bit sprites, this is similar but the sprites are actually 8-bit ones. They can still be defined in 8 bit index values however these values' 4 top bits will get chopped off and replaced by the optional offset. Since calculating and/or anticipating and properly structuring your palettes for such a use can be a large hassle; it's good practice if you want to use this feature to define your sprite values from 0 to 15 and set the offset to adjacent sets of 16 colours. This way in a potential future version of *NextBASIC* that supports native 4-bit sprites, you won't have to change pattern definitions at all.

## Relative sprites

Sprites can be grouped together to form *composite* or *unified* sprites. Each such grouping consists of a single *anchor* sprite which is the sprite with the lowest *id* in the grouping, followed by any number of *relative* sprites, with sprite *id*s following the anchor sprite in sequence.

When an anchor sprite moves or becomes invisible, all the associated relative sprites also move or become invisible. It is also possible for individual relative sprites to be made invisible or visible. The rule is that a relative sprite is only visible if its own visibility flag is set and the visibility flag of the associated anchor sprite is set.

To define a relative sprite, simply specify its sprite id as a negative number for example specifying:

SPRITE -1,....

defines sprite 1 as relative to the preceding anchor sprite, with id: 0.

Any number of relative sprites can follow an anchor sprite.

The x and y coordinates specified in SPRITE commands for relative sprites are not actual coordinates, but signed integer offsets in the range -128 to +127 from the coordinates of the anchor sprite. This establishes how close the anchor and the relatives are; in other words they don\t have to touch each other; only when we want to create something visibly bigger looking like one single thing on screen!

Additionally, if the pattern relative flag is set for a particular relative sprite, its pattern number is added to the pattern number from the anchor sprite – wrapping round if the sum exceeds 64.

Using this, it is easy to animate an entire composite/unified sprite simply by changing the pattern of the anchor sprite. This is extremely similar to our small animation example before.

In the same vein, if the *palette relative* flag is set for a particular relative sprite, its palette offset is added to the palette offset from the anchor sprite (wrapping round if the sum exceeds 16).

## Composite vs Unified sprites

The type of a grouping of sprites is determined by the *type* flag of the anchor sprite: composite or unified. The distinction between them is simple:

For composite sprites, the remaining sprite parameters (rotation, x/y mirrors and x/y scaling) are independent for each relative sprite. This allows creation of a composite sprite where individual relative sprites can be rotated etc for animation purposes whereas for unified sprites, the rotation and x/y mirrors of the relative sprites are relative to that of the anchor sprite.

Therefore, when the rotation or x/y mirrors of the anchor are changed, all the relative sprites rotate or reflect about the anchor. The same goes for the x and y scaling of the indi-

vidual relative sprites in the case of unified sprites; it is completely ignored, thus allowing the entire grouping to be scaled just by changing the scaling of the anchor[3].

## Batching

The standard SPRITE command normally has immediate effects to what is displayed on the screen. However, it is also possible to place *NextBASIC* into *batching mode*.

In this mode, SPRITE command has no immediate effect, but the changes specified are remembered. When all the required changes have been made using multiple SPRITE commands, they can all be applied to the screen at once, giving a more synchronised look to your game and fewer screen tears.

To control batching the following commands are available in the order one would use them:

SPRITE STOP

which enables batching mode and turns off the immediate sprite screen updates

SPRITE MOVE

This command sends all outstanding sprite changes to the hardware immediately (ergo displays all the animation effects and movement that was pending while in batching mode).

SPRITE MOVE INT

The same as above but waiting for the 50 Hz / 60 Hz interrupt to occur. This is useful in order to synchronise the sprite movement to the framerate (making for a smoother animation) and finally

SPRITE MOVE INT *y*

which works in the same way as SPRITE MOVE INT, except that changes are not sent to the hardware until after the TV scanline corresponding to sprite coordinate *y*. This avoids flicker by making sure that sprite changes do not happen on screen while the display is midway through displaying the current sprite(s). Finally

SPRITE RUN

disables batching mode and turns the immediate screen updates back on. This is also done by the SPRITE CLEAR command we saw earlier but without the destructive effects.

## Automatic sprite movement

As we saw above, moving a sprite can be laborious. In order to reduce the amount of work a *NextBASIC* program needs to do to animate and move sprites, commands are provided to allow some or all of this work to be done automatically whenever a SPRITE MOVE command is issued. Any sprite can have automatic movement or animation applied to it, and the standard SPRITE command can still be used to perform any other changes when they are needed.

The main command used to set up automatic sprite movement is the SPRITE CONTINUE command:

SPRITE CONTINUE *s*, [*x1* [TO *x2*]] [STEP *xs*] [RUN|STOP], [*y1* [TO *y2*]] [STEP *ys*] [RUN|STOP], [*p1* [TO *p2*]],[*f*], [*r*], [*d*]

Although looking somewhat daunting at first, SPRITE CONTINUE is quite easy to master regardless of its numerous options. As apparent by the brackets, each parameter (or sub-clause of a parameter) is optional. If not specified, the previous value will be retained.

Movement in the *x-direction* is specified with:

3 *This scaling also applies to the relative x/y coordinate offsets*

*x1*: minimum value for *x-coordinate*
*x2*: maximum value for *x-coordinate* (or assumed to be equal to *x1* if not provided)
*xs*: signed step in pixels (between **-127...+127**) for every horizontal move
and
**RUN**: indicates movement in the *x-direction* is initially **on** or
**STOP**: indicates movement in the *x-direction* is initially **off**

The parameters are equivalent for the *y-direction* and specified with:

*y1*: minimum value for *y-coordinate*
*y2*: maximum value for *y-coordinate* (or assumed to be equal to *y1* if not provided)
*ys*: signed step in pixels (between **-127..+127**) for every move
**RUN**: indicates movement in the *y-direction* is initially **on**
**STOP**: indicates movement in the *y-direction* is initially **off**

Pattern animation is specified with:

*p1*: minimum value for sprite pattern
*p2*: maximum value for sprite pattern (or *p1* if not specified!)

while movement rates are controlled with:

*r*: rate at which sprite moves/animates (**0-255**) where...
**0**=on every **SPRITE MOVE** command
**1**=skip 1 **SPRITE MOVE** command after moving
**2**=skip 2 **SPRITE MOVE** commands after moving
etc

*d*: delay before initial movement (**0-255**) where:
**0**=move on the first **SPRITE MOVE** command
**1**=skip 1 **SPRITE MOVE** command before the first move
**2**=skip 2 **SPRITE MOVE** commands before the first move
etc

The flags parameter *f* is an 8-bit mask (which as seen before is best specified with **BIN** or **@**):

bits **1:0** define the behaviour when the *x,y* limits are reached:
    **00** = reflect this direction
    **01** = stop this direction, start other direction
    **10** = stop this direction
    **11** = stop completely and make sprite invisible
bit **2** flips the *Y-mirror flag* when *y* limits are reached
bit **3** flips the *X-mirror flag* when *x* limits are reached
bit **4** controls the behaviour of pattern change:
    **0** = cycle upwards, wrapping back to lower limit
    **1** = bounce between lower and upper limits
bit **5** if set, sprite is disabled when its pattern reaches limits
bit **6** if set, updates the pattern even when sprite is stationary and finally
bit **7** if set, *X-mirror/Y-mirror/rotation flag* are set according to the direction of travel
    (this bit overrides bits **2** & **3**)

The initial position, pattern and other details of the sprite are determined by the last standard **SPRITE** command. If these values are outside the maximum/minimum ranges, then (depending upon the specified *step* and **RUN|STOP** status) they will gradually change until they fall within the max/min range.

If automatic movement is specified for an *anchor* sprite then, the entire *composite* or *unified* sprite will be automatically moved.

Automatic movement, can also be specified for individual relative sprites if this is desired (Since the parameter *s* is always **positive** for the **SPRITE CONTINUE** command, but the sprite remains relative if specified as such in the last standard **SPRITE** command).

This would be done to animate the relative sprites separately, so that one relative sprite might animate whilst the others remain the same (for example). However, it can also be used to automatically move a relative sprite around within a composite/group sprite. The main restriction here is that the movement limits are unsigned, so this works best for relative sprites with positive offsets (add **256** to negative offsets when specifiying them in a movement range).

Automatic movement can be temporarily suspended for particular (or all) sprites if so desired by using:

SPRITE PAUSE *s1* [TO *s2*]

which turns off automatic movement for a single sprite *s1* or a range (*s1 – s2)* of sprites. Said suspension is lifted by using:

SPRITE CONTINUE *s1* [TO s2]

which restarts automatic movement for a single or a range of sprites as above.

## Sprite functions

In order to return details about sprites, and to make collision detection checks, several functions are provided[4]. These are:

SPRITE *s*

which shows if sprite *s* is visible. Returns **1** (true) if sprite is visible or **0** (false) if not.

SPRITE CONTINUE s

Returns a bitmask describing the automatic movement enabled for sprite *s*:
bit **0**: set if automatic movement is enabled
bit **1**: set if currently moving in the *Y axis*
bit **2**: set if currently moving in the *X axis*

Note that if bit **0** is set but neither bits **1**or **2** are set, that means that only the pattern is being animated.

SPRITE AT(*s,c*)

Returns a coordinate or other movement-related value for the sprite:

| | |
|---|---|
| SPRITE AT(*s*,**0**) | returns *x coordinate* |
| SPRITE AT(*s*,**1**) | returns *y coordinate* |
| SPRITE AT(*s*,**2**) | returns *pattern number* |
| SPRITE AT(*s*,**3**) | returns *x step* |
| SPRITE AT(*s*,**4**) | returns *y step* |
| SPRITE AT(*s*,**5**) | returns *delay* before the sprite next moves[5] |

One of the most labour-intensive game programming tasks is trying to figure out when two sprites have collided. NextBASIC provides that information with

SPRITE OVER(*s1, s2* [TO *s3*] [,*overlapX* [,*overlapY*] ])

which performs a bounding-box collision detection between sprite *s1*, and a single other sprite *s2* or a range of sprites *s2...s3*.

Two optional acceptable overlaps (in pixels) can be provided in *overlapX* and *overlapY*. If *overlapX* is not present, **0** (no overlap) is used. If *overlapY* is not present, then the value of *overlapX* is used. Overlaps should be **0...7** for an unscaled sprite *s1*, or **0...15** for a 2x scaled sprite etc. Overlaps allow for some flexibility before declaring a collision especially since sprite patterns do not always extend to the boundaries of the sprite bounding box (16 x 16)

4 *All these functions are available in the standard expression evaluator and the integer expression evaluator*
5 *A returned value of **0** means the sprite will move on the next **SPRITE MOVE** command.*

The function returns 0 (false) if there is no collision or the number of the colliding sprite (*s2...s3*) if there was a collision.

NOTE: If the colliding sprite's id is 0 then 128 is returned.

Any relative sprites following *s2* or *s3* will also be checked, until the next anchor sprite that is not in the specified range.

## Scrolling

The last method of animation is by using the in-built *hardware scrolling* capabilities of the ZX Spectrum Next. As you will find out in *Chapter 22*, all layers can be scrolled either in full or within a clipping window (see *Chapter 16 – Graphics*). *NextBASIC* provides access to *hardware scrolling* via the LAYER AT command. Its syntax is as follows:

LAYER AT *x,y*

which moves the current layer to the offset defined by the coordinates *x* and *y*. According to which side we're moving to, the existing graphics on that side get wrapped around the opposite side. Let's demonstrate using one of the images we generated earlier while doing frame-based animation:

```
10 LAYER 2,1:CLS
20 LOAD "ANIM0.SL2" LAYER: PAUSE
    0: REM Hasta la vista Kev!
30 FOR %x=0 to 255
40 LAYER AT %x,%0
50 NEXT %x
60 LAYER AT 0,0: LAYER 2,0:LAYER 0
```

Once you RUN the above, you'll see an image racing towards the left side of the screen so fast it may even be unusable for anything other than a simple effect. Running it at 3.5MHz you will see a very smooth movement which shows how efficient hardware scrolling is on the ZX Spectrum Next.

If you want to reverse the effect and make the screen move towards the right you will need to change line 40 to:

```
40 LAYER AT %255-x,0
```

If we borrow a bit from the sprite example, we can even introduce a SIN function to make the screen appear like it's bouncing from left to right and top to bottom and vice-versa.

By itself, the LAYER AT keyword doesn't do much other than roll a screen around; with the combination however of layer clipping windows and background updating of the shadow screens (See *Chapters 22* and *23* as well as *Chapter 16*), you can produce a scrolling effect of very large landscapes. If you combine this with specially crafted screens that can repeat themselves at infinitum then you have the basics for every side scrolling game ever made!

## The Copper

While not strictly an animation aid, the Copper is a hardware module of the ZX Spectrum Next that can definitely be used for, among other things, animation. The Copper runs in parallel and independently from the main Z80n processor and is dedicated to writing Next Registers (NexREG) at specific points on the display. The name derives from “co-processor” and was first seen in the Amiga computer which had a similar function. The Copper, essentially maintains a list of instructions that consists of only two commands; WAIT and MOVE. This simple control allows updating of Next registers at regular times, synchronised to points when the display is updated on the screen. The Copper system can therefore be used to send audio samples to the ZX Spectrum Next's digital audio hardware,

make fast colour changes to get sky effects, change layer priorities, enable or disable screen modes etc. all that from a simple list of commands.

On older Spectrum models, you would have needed some very clever use of the Interrupt system to do these sort of tricks with some being completely impossible or just too slow to be of any practical use. Even with the ZX Spectrum Next's ability to generate interrupts on each raster line, setting that up (especially in *NextBASIC*) and then trying to get the timing right for nice clean effects is very complicated (or impossible) and yet simple to accomplish by using the Copper.

We'll jump ahead a bit and introduce a special command; REG (which will be covered in full in *Chapter 22*). For now take REG n,v to be the same as OUT 9275, n: OUT 9531,v. Let's see our example:

```
  10 BORDER 0: PAPER 0: INK 7:CLS
  20 REG 98,0: REM make sure Copper is
     stopped
  30 REG 97,0
  40 REM Select the Copper data
     register
  50 FOR x=0 TO 6: REM Increase this
     if you add more data lines.
  60 READ m,l
  70 REG 96,m: REG 96,l: REM write the
     Copper list from DATA statements
  80 NEXT x
  90 REG 97,0: REM low part of address
 100 REG 98,%@11000000: REM high part
     of address and start Copper,
     repeat on VBlank
1000 DATA 128+(45*2),0:
     REM WAIT for line zero horizontal
     45
1010 DATA 64,16,65,BIN 11100000:
     REM WRITE Palette Index 16 (Paper
     and Border), then WRITE RED
1020 DATA 128+(45*2),100:
     REM WAIT for line 100 horizontal
     45
1030 DATA 64,16,65,BIN 00000000:
     REM WRITE Palette Index 16 and
     WRITE contents back to BLACK.
1040 DATA 128+1,128
1050 REM Last line waits for a bit of
     the screen that does not exist
     1*256+128 = 386 (STOP)
```

You can try changing the BIN statements in lines 1010 and 1030 to use different colours – this is the 8 bit Palette value so RRRGGGBB

Now remember this list is still running in the background but, it is changing ULA palette 0 paper colour. *NextZXOS* uses palette 1 so you do not see it when editing *NextBASIC*. Just type CLS and you will see that it comes back until you press a key!

WAIT commands (where the top bit is 1 i.e. bytes >128) will pause processing until a certain point on the display (to a fixed resolution).

MOVE commands (where the top bit is 0 i.e. bytes <128) will take a given value and put it in the numbered register.

You can have up to 1024 commands which can repeat or stop at any point by WAITing for a non existent line I.e. >311 which works at both 50 and 60 Hz. So there is loads of room for creativity and invention.

Only the lower 128 Next registers can be written but, this is not an issue as the registers above 127 are mainly used for the accelerator and the Expansion Bus.

Register 96 (60h) is the data port to write the instructions. They are two bytes long so you need to write them in pairs with the most significant byte first – not the usual Z80 way but, needed for the way the system works.

Register 97 and 98 (61h and 62h) are the controls; the first is the low 8 binary bits of the address to WRITE the instructions, the second contains the bits to control the mode and the top bits of the instruction address. If you change to mode 01b (from another mode like 00b PAUSE/STOP) this also resets where the Copper begins to READ its instructions from back to instruction 0 – in all other cases it will carry on from where it left off last time.

The Copper sees the screen starting from the top left pixel of the display area of the screen, this is 0,0. After 32 horizontal values (every 8 pixels) you have the right border, then you have a gap (count of 12) which is where, on an old TV, the spot would be flying back over to the left, then you have the right hand border of the next horizontal line.

Note: This zero point is also where the screen "dot" will be when the first *Raster Line Interrupt* occurs. Do not confuse this with normal interrupts on the system which occur in the top left of the whole screen as it is displayed on a monitor or TV. That is actually somewhere in the middle of the bottom right of the Copper view of the screen shown in the diagram below. Exactly at raster line 224 at 60Hz or 248 at 50Hz.

Finally when it gets to the bottom of the screen it has the border and then a blank period (8 lines) while the old spot was running back to the top of the screen, then you have a number of lines in the top of the screen area to play with (56 at 50Hz or 32 at 60Hz). To see this change line 1000 for **DATA 128+(45*2),200** and line 1020 for **DATA 128+(45*2)+1,45**. Remember: **1*256+45 = 301**.

This diagram will hopefully help to visualise that:

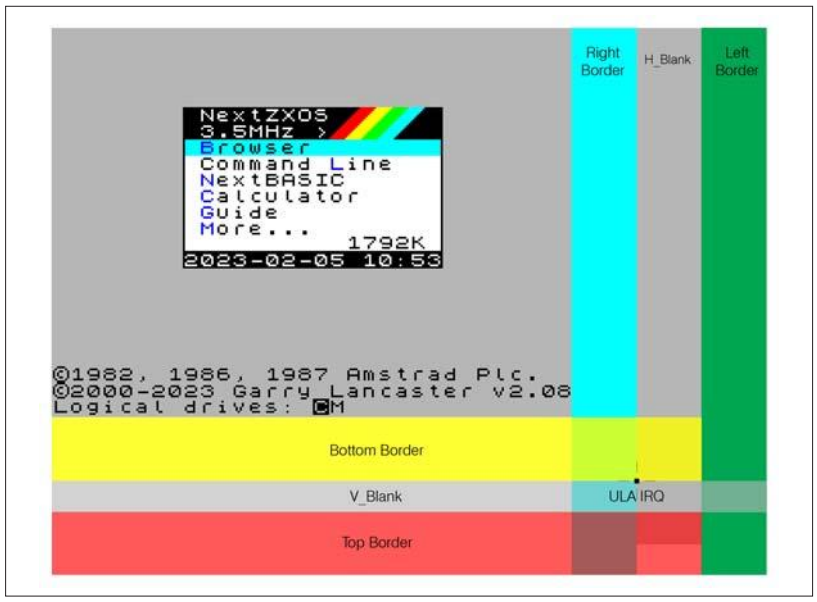



*Fig. 16 – Copper operation*

If you MOVE 0,0 (i.e. Write something to a Read Only Next Register like Register 0) then the Copper does nothing for a short duration (a NOP in Z80 terms) so you can wait for a more accurate moment to overcome the fact you only have 55 horizontal positions to wait for i.e. every 8 pixels on the screen.

You can write to the Copper as it is running because it keeps a separate track of its READ instruction address to the address you are using to WRITE.

WARNINGS:

Be careful as the *NextZXOS Screensaver* uses whatever palette is in place so if you have any border effects running they will still be visible and could cause a CRT screen to experience a burn-in effect. This is worth bearing in mind if you are writing software not to leave static images around too long!

If you try to write to a Next Register at the same time as the Copper then this might cause a conflict – don't worry; the Copper will win and the display will be OK but, your program command may fail.

So some care is needed to manage the two systems. Turning off the Copper while you make Next Register affecting changes in *NextBASIC* is a good idea. That includes things like the PALETTE command for example. If you are using machine code you will need to use some form of flag and remember what the Copper might be doing at a specific time.

In the above program for example, it is possible the Copper STOP in the first two lines will fail if you run it a second or third time to change the colour and will not reset the write address, so you will write after the list already there and your new one will never be reached. You could get around that by repeating the first two lines as it is unlikely to fail twice so shortly after the last attempt and has no effect if it does run twice.

## Exercises

1. Write a procedure to write a STOP command twice in a row so that you can make sure the Copper is stopped when you need to in your programs.

2. Draw a Spectrum Flash on the right hand side border by changing the palette colour five times – make sure the last time is back to your real paper/border colour. Hint you can use one or more WRITE 0,0 as a very short delay.

3. Write a program that controls two spaceships using the sprite defined, one going horizontally, while the other vertically on the screen

4. Enhance the above program with a memory based *Layer 2* animation running in the background

# Chapter 18 – Sound and Music

Unlike its predecessors, your ZX Spectrum Next doesn't fare poorly in the audio capabilities department. From simple beeps and clicks, to complex compositions using its in-built 3 Programmable Sound Generators (*PSGs*) and full-fledged digital audio output, sound can accompany almost every program you write or software you will load. Sound is output in stereo from both the *digital video port* and an analogue 3.5mm jack output present on the back of the machine. Additionally, there is the possibility of an on-board *piezo speaker* (sold separately).

## Basic sounds with the BEEP command

The easiest way to create sounds (and the only method that works on all ZX BASIC versions including *NextBASIC*) is by using the **BEEP** statement:

**BEEP** *duration, pitch*

where, as usual, *duration* and *pitch* represent any numerical expressions. The *duration* is given in *seconds*, and the *pitch* is given in *semitones* above *middle C*. For notes below *middle C* we use negative numbers.

Here is a diagram to show the pitch values of all the notes in one *octave* on the piano:

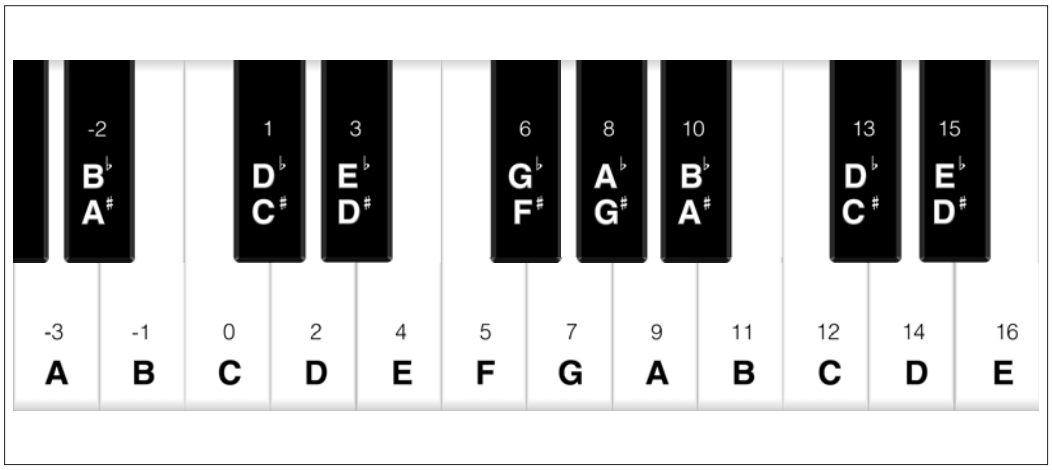



*Fig. 17 – Pitch/note equivalents*

To get higher or lower notes, you have to add or subtract **12** for each *octave* that you go up or down.

If you have a piano in front of you when you are programming a tune, this diagram will probably be all that you need to work out the pitch values. If, however, you are transcribing straight from some written music, then we suggest that you draw a diagram of the stave with the pitch value written against each line and space, taking the key into account.
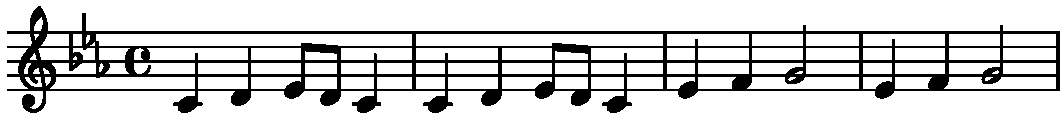
For example, type:

```
10 PRINT "Frere Gustav"
20 BEEP 1,0: BEEP 1,2: BEEP .5,3: BEEP
   .5,2: BEEP 1,0
30 BEEP 1,0: BEEP 1,2: BEEP .5,3: BEEP
   .5,2: BEEP 1,0
40 BEEP 1,3: BEEP 1,5: BEEP 2,7
50 BEEP 1,3: BEEP 1,5: BEEP 2,7
60 BEEP .75,7: BEEP .25,8: BEEP .5,7:
   BEEP .5,5: BEEP .5,3: BEEP .5,2: BEEP
   1,0
70 BEEP .75,7: BEEP .25,8: BEEP .5,7:
```

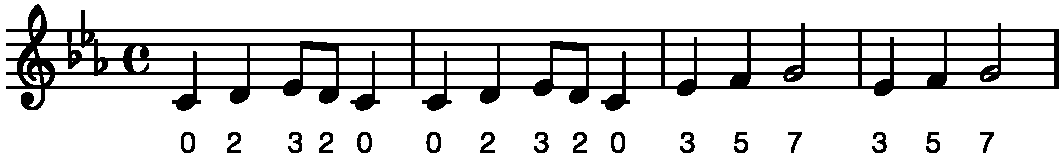
```
      BEEP .5,5: BEEP .5,3: BEEP .5,2:
      BEEP 1,0
80 BEEP 1,0: BEEP 1,-5: BEEP 2,0
90 BEEP 1,0: BEEP 1,-5: BEEP 2,0
```

When you run this, you should get the funeral march from Mahler's first symphony, the bit where the goblins bury the US Cavalry man.

Suppose for example that your tune is written in the key of *C minor*, like the Mahler above. The beginning looks like this:



and you can write in the pitch values of the notes like this:




We have put in two ledger lines, just for good measure. Note how the **E flat** in the key signature affects not only the **E** in the top space, flattening it from **16** to **15**, but also the **E** on the bottom line, flattening it from **4** to **3**. It should now be quite easy to find the pitch value of any note on the stave.

If you want to change the key of the piece, the best thing is to set up a variable **key** and insert **key+** before each pitch value: thus the second line becomes:

```
20 BEEP 1,key+0: BEEP 1,key+2: BEEP .5,
   key+3: BEEP.5,key+2: BEEP 1,key+0
```

Before you run a program you must give **key** the appropriate value – **0** for **C**, **2** for **D**, **12** for **C** an *octave up*, and so on. You can get the computer in tune with another instrument by adjusting **key**, using fractional values.

You also have to work out the durations of all the notes. Since this is a fairly slow piece, we have allowed one second for a *crotchet* and based the rest on that, half a second for a **quaver** and so on.

More flexible is to set up a variable **crotchet** to store the length of a **crotchet** and specify the durations in terms of this. Then line 20 would become:

```
20 BEEP crotchet,key+0: BEEP crotchet,
   key+2: BEEP crotchet/2,key+3: BEEP
   crotchet/2,key+2: BEEP crotchet, key+0
```

(You will probably want to give **crotchet** and **key** shorter names.)

By giving **crotchet** appropriate values, you can easily vary the speed of the piece.

When using **BEEP**, one must remember that via *NextBASIC* we can only produce one tone per unit of time since this is done via the *CPU*, therefore you are restricted to unharmonised tunes. If you want harmonies, you should either use the **PLAY** command described in the following section or program the computer in *Machine Code*. Further-

more, since tone generation via the *CPU* is an exclusive task, you cannot do anything else on or off screen while the sound is playing, so in order to perform other functions while sound is generated by using the *CPU*, you will also have to program in *Machine Code*, or –assuming you have the *Accelerated* version or a *Pi Zero* installed– use the audio playback facilities described in the last section of this chapter (the latter working independently of whatever the ZX Spectrum Next is doing).

Try programming tunes in for yourself – start off with fairly simple ones like *Three Blind Mice*. If you have neither piano nor written music, find a very simple instrument like a tin whistle or a recorder, and work the tunes out on that. You could make a chart showing the pitch value for each note that you can play on this instrument.

Type:

```
FOR n=0 TO 1000: BEEP .5,n:
NEXT n
```

This will play notes as high as it can, and then stop with error report **B Integer out of range**. You can print out **n** to find out how high it did actually get.

Try the same thing, but going down into the low notes. The very lowest notes will just sound like clicks; in fact the higher notes are also made of clicks in the same way, but faster, so that the human ear cannot distinguish them.

Only the middle range of notes are really any good for music; the low notes sound too much like clicks, and the high notes are thin and tend to warble a bit.

Type in this program line:

```
10 BEEP .5,0: BEEP .5,2: BEEP .5,4:
   BEEP .5,5: BEEP .5,7: BEEP .5,9:
   BEEP .5,11: BEEP .5,12: STOP
```

This plays the scale of *C major*, which uses all the white notes on the piano from *middle C* to the *next C* up. The way this scale is tuned, is exactly the same as on a piano, the so-called *even-tempered tuning* because the pitch interval of a *semitone* is the same all the way up the scale. A violinist, however, would play the scale very slightly differently, adjusting all the notes to make them sound more pleasing to the ear. He can do this just by moving his fingers very slightly up or down the string in a way that a pianist can't.

The *natural scale*, which is what a violinist would play, comes out like this:

```
20 BEEP .5,0: BEEP .5,2.039: BEEP .5,
   3.86: BEEP .5,4.98: BEEP .5,7.02:
   BEEP .5,8.84: BEEP .5,10.88:
   BEEP .5,12: STOP
```

You may or may not be able to detect any difference between these two; some people can. The first noticeable difference is that the third note is slightly flatter in the *naturally tempered scale*. If you are a real perfectionist, you might like to program your tunes to use this natural scale instead of the even-tempered one. The disadvantage is that although it works perfectly in the *key* of *C*, in other *keys* it works less well – they all have their own natural scales – and in some *keys* it works very badly indeed. The *even-tempered scale* is only slightly off, and works equally well in all keys.

This is less of a problem on the computer, of course, because you can use the trick of adding on a variable **key**.

Some music – notably Indian music – uses intervals of pitch smaller than a *semitone*. You can program these into the **BEEP** statement without any trouble; for instance the *quartertone* above *middle C* has a pitch value of **.5**.

You can make the keyboard beep instead of clicking by:

```
POKE 23609,255
```

The second number in this determines the length of the beep (try various values between 0 and 255). When it is 0, the beep is so short that it sounds like a soft click.

## Enhanced Sound and Music with PLAY

When using *NextBASIC*, you have two different ways to make music and sound effects. You can still use the BEEP command (as discussed above) but you also have access to the PLAY command which allows you to make much more sophisticated music with up to *nine* notes playing at once. It also gives you more control over the sound of each individual note than is possible using BEEP.

Making music and sound effects with PLAY is simple. You just type in the series of notes that make up a tune, then ask the ZX Spectrum Next to PLAY them. You can also include instructions that tell your machine what sort of tone you want for the sound. Please note that case is important when typing in the string expressions in the examples ie. ga should not be typed as Ga, gA or GA.

To hear some of the wide range of sounds that you can make, type in one of the two programs below, RUN it, then try the other example. Don't worry if the program lines look complicated, they are explained in detail later.

Music:

```
10 b$="O4(CDEC)(5EF7G)(3GAGF5EC)
   5Eb7E9EbE"
20 PLAY "T180O6(CDEC)(5EF7G)(3GAGF5EC)
   5Cg7C9CgC",b$,"O3(7CG)(7CG)(7CG)
   5GD7G9GDG"
```

Sound Effects:

```
10 a$="M8UX350W507(((C)))": PLAY a$ :
   PAUSE 25
20 PLAY "M56UX5000W103(((C)))": PAUSE 25
30 a$="M56W201N8C" : PLAY a$ : PAUSE
   25
```

## Using the PLAY command

In the examples above, you will see that each time the PLAY command appears, it is followed by up to *nine* different parameters in the form of either *string variables*, *string literals* or a combination of both in a statement like:

PLAY *P1C1,P1C2,P1C3,P2C1,P2C2,P2C3,P3C1,P3C2,P3C3*

where P*x*C*y* are strings that refer to the *PSG* (P) number (*x*) (1 to 3) and *channel* (C) number (*y*) (1 to 3). The order of these is specific and each PLAY command must have the full complement if you require all the channels to reproduce a sound. You cannot issue two or more PLAY commands to control individual PSGs as each PLAY statement sends a batch of instructions to the audio hardware. If you wish one or more channels to be silent you should replace them with the empty string "". As we will examine below, the strings contain all the information to tell your ZX Spectrum Next which sounds to make.

As we discussed, PLAY controls *nine* separate sound *channels* over the *3* available *PSGs*, each called A, B, and C.

In the *Music* example given above, "T180O6(CDEC)(5EF7G)(3GAGF5EC)5Cg7C9CgC" tells *channel* A of *PSG1* to play the melody line, b$ tells *channel* B of *PSG1* to play a harmony, and "O3(7CG)(7CG)(7CG)5GD7G9GDG" tells *channel* C of *PSG1* to play a bass part. In the *Sound Effects* example, only one *noise* is used at a time (although up to *nine*

can be), so each one is in channel A of *PSG1* and the command is simply PLAY a$ – or (as seen in line 20) PLAY "M56UX5000W1O3(((C)))".

In fact any of the *channels* can produce either a *musical tone* or *noise* or even nothing at all, so you can mix sound effects in with your music (see *Channel selection* later on).

## Constructing strings

Composing music and sound effects in *NextBASIC* is just a matter of creating strings containing the information you want. Try this – very simple – example, which plays just one note – an A.

```
a$="a": PLAY a$
```

Any music program using PLAY will generally use *string variables* rather than literals to tell it what to play, as you can see by looking at the earlier examples. The more complex, or longer, the piece and the more complicated sound, the more complex the strings become as obvious from the increased complexity of the examples above.

Any *musical sound* has a *pitch* and *duration*. It also has a *volume* and *timbre*. The strings in the earlier examples contain information about all of these. The summary below lists each possible command, and they are explained in detail opposite.

## PLAY command summary

This is a brief list of the commands which can be contained in a PLAY string. Note that all letters except note names must always be in capitals.

| String entry | Function |
|---|---|
| c-b or C-B | Gives pitch of note within current octave range |
| $ | Flattens note following it |
| # | Sharpens note following it |
| Ox | Sets octave range x (0 to 8) |
| 1-12 | Sets duration of note |
| & | Denotes a rest |
| N | Separates two numbers |
| Vx | Sets volume to x (0-15) |
| Wx | Sets volume effect to x (0-7) |
| U | Turns on volume effect in the current channel |
| Xx | Sets duration of volume effect to x (0-65535) |
| Tx | Sets tempo to x (60-240) bpm |
| ( ) | Enclose repeated phrase |
| ! ! | Enclose a comment |
| H | Halts a PLAY command |
| Mx | Selects channel and sets type to x (1-63) |
| Yx | Turns on MIDI channel x (1-16) |
| Zx | Sends x as a MIDI patch |
| L | Restricts output from current PSG to Left Speaker Only |
| R | Restricts output from current PSG to Right Speaker Only |
| S | Restores stereo mode to current PSG |

*Table 10 – **PLAY** commands*

## Setting the pitch

As you saw above, you set the pitch of any note by giving its musical name – eg. C E G. *Sharp* notes are prefixed by # (eg #C) and flat notes by $. A *two-octave range* in the *key of C*, which use the letters c to b for the notes in the lower *octave* and C to B in capitals for the

higher one are available at any moment. Any number of notes within these two *octaves* can be played one after another, for example:

```
10 a$="cfedafgCFEDAFGCC"
20 PLAY a$
```

If you want to span more than just two *octaves*, you can change the overall pitch of the *channel* playing by using the *octave command* O followed by a number from 0 to 8. If you do not specify an *octave* (as in the example above), this defaults to 5 (the range containing *middle C*). The *octave command* remains in force for all notes following it until a new *octave* command is given.

This program lets you hear the same tune played in a higher *octave* (just add the O7 to your earlier program):

```
10 a$="O7cfedafgCFEDAFGCC"
20 PLAY a$
```

Try changing the *octave* number progressively to hear the full pitch range which your ZX Spectrum Next's *PSGs* can produce.

Since each pitch range covers two *octaves*, two adjacent ranges overlap. For example, the high part of O4 contains the low part of O5 (see *Figure* below). The following diagram shows how you can create different notes using the PLAY *octave command*. As mentioned previously, the command O followed by a number from 0 to 7 sets the current *PSG* to a range of two *octaves* beginning with a *C*. The diagram shows the complete range of notes covered by O3, O4, and O5. Adjacent *octave* ranges overlap, so the same notes appear in the upper part of one range and the lower part of another. Individual notes within an *octave* range are set by using the letters c to b in lower case for the lower notes and C to B in capitals to give the notes in the upper *octave*. Placing a # before any note letter gives a sharp note – a $ flattens it.

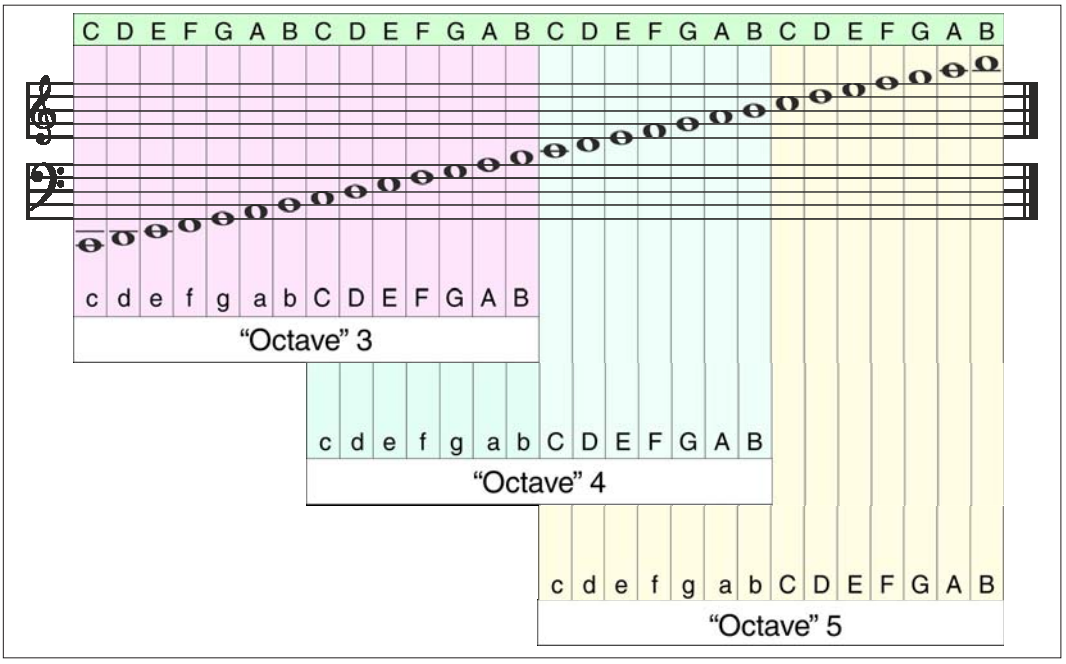



*Fig. 18 – Octaves and Pitch values for making music with* ***PLAY***

## Note duration

If you do not specify the length of each note, they will all be played at the same length (as *crotchets*) as in the examples above. You can fix the length of any note or series of notes by prefixing it with a number from 1 to 12. This program lets you hear the different note du-

ration with numbers from 1 to 9 (there is a reason for the maximum number being 9 in this example as you will see in the table below).

```
10 a$="1C2C3C4C5C6C7C8C9C"
20 PLAY a$
```

The PLAY command supports *9* standard musical durations: from a *semiquaver* (*sixteenth* note) to a *semibreve* (*whole* note) of the time signature. There are three extra duration values which denote *triplet* notes (three notes played in the time normally used for two): from a *triplet semiquaver* (*triplet sixteenth*) to a *triplet crotchet* (triplet quarter). While the first 9 values are set and apply to all the notes that follow, a triplet duration value (10-12) only applies to the next 3 notes that will follow it in the string. For example:

```
10 PLAY "11ACE"
```

plays a *triplet quaver* of A, C and E. The following table lists the note duration values and their musical term equivalent.

| Value | Note name (Standard) | Note name (British) | Musical notation |
|---|---|---|---|
| 1 | Sixteenth | Semiquaver |  |
| 2 | Dotted sixteenth | Dotted semiquaver |  |
| 3 | Eighth | Quaver |  |
| 4 | Dotted eighth | Dotted Quaver |  |
| 5 | Quarter | Crochet |  |
| 6 | Dotted Quarter | Dotted Crochet |  |
| 7 | Half | Minim |  |
| 8 | Dotted Half | Dotted Minim |  |
| 9 | Whole | Semibreve |  |
| 10 | Triplet sixteenth | Triplet semiquaver |  |
| 11 | Triplet eighth | Triplet quaver |  |
| 12 | Triplet quarter | Triplet crotchet |  |

*Table 11 – Note duration values*

Additionally there is also the ability to insert moments of silence (or *rests* as they're called in music terminology) denoted by the ampersand symbol (&). *Rests*, last as long as the current note playing. For example:

```
10 PLAY "7A&B&C&D&E"
```

is five *minims* with equal (*minim*-length) silence durations between them.

*Tied* notes can be indicated by giving the two note durations connected by an *underscore character* (_) and the note name, eg.:

```
10 PLAY "3_5A"
```

The second note duration you give will also apply to any following codes until you give another duration code.

## The N Command

In some of the examples you will see the letter N used to introduce a series of notes within the string:

```
PLAY "O7N1CDE"
```

N is used in cases where two sets of numbers would otherwise clash. In the example above, O is set to octave 7, then a series of notes is given, starting with the duration code 1. Without the N code, *NextBASIC* would read the octave code as 71 – obviously not what was intended!

## Note volume

The overall volume of the sound is controlled by the volume setting of your display or amplifier. You can control, however, the volume of individual notes and phrases within the tune by using the V command. V followed by a number from 0 to 15 sets the note(s) that follow to a constant volume level. The lower the number, the *quieter* the sound, with V0 being completely silent (V0 is a useful way of stopping one *channel* playing while others continue). V15 is the maximum possible value and will be used automatically by *NextBASIC* if you do not specify a level.

The low volumes are very quiet and you will normally use 10 to 15 unless you are outputting to an amplification system. Try running this program:

```
10 a$="V10cdefgabCDEFGAB"
20 PLAY a$
```

Now try changing the number after the V to a new value to hear the difference.

## Volume effects

Instead of you just setting each note to a fixed volume, PLAY also lets you change the volume of the sound while it is playing. For example, you can make a note start suddenly and then die away (like a piano) or make a sound effect rise and fall in volume (like a steam train).

This effect is controlled by the letter W which can be included in any of the strings controlled by the PLAY command. You must also include the letter U in each string where you want to use the effect. You cannot use it if the string already has a volume setting (if it contains a V) – the volume command will override the effect.

The W must be followed by a number from 0 to 7 which controls how the sound builds up (called *attack*) or falls off (called *decay*). *Table 13* that follows, shows the full range of numbers and what they do together with a visual representation of the volume effect applied to the sound playing:

This program plays the same note with each effect in turn to let you hear what they sound like:

```
10 a$="UX1000W0C&W1C&W2C&
   W3C&W4C&W5C&W6C&W7C"
20 PLAY a$
```

Notice the U to turn on the effect, then the series of W numbers.

There is one other new command used here, the letter X. This can be followed by a number from 0 to 65535 to set the length of the sound effect – the larger the number, the longer the effect lasts.

The X command is not mandatory. If you choose not to include one, *NextBASIC* will automatically choose the longest. In general, repetitive effects (W4 to W7) are more effective

with short settings, eg X300. *Single-shot* effects (W0 to W3) need a longer period, eg X1000. Try changing the value after X in the program above to hear the difference.

## Tempo

The speed (*tempo*) at which a piece of music is played can be set with the command T followed by the number of *crotchet beats per minute* (bpm) in the range 60 to 240. The command controls the speed at which all notes are played, but can only be included in *channel* A of *PSG1* (the first string after the PLAY command) otherwise it is ignored, eg:

```
10   a$="T180cdefg"
20 PLAY a$, "T120CDEFG"
```

will play *octave chords* but at 180bpm as the second setting is ignored. If no *tempo* is specified, the music will be played at 120 bpm.

## Repeated phrases

Any musical phrase can be repeated by enclosing the appropriate string or part of a string in parentheses. For example:

```
10 PLAY "abC(DEFG)"
```

will repeat the last four notes. If there is an unequal number of parentheses, the phrase will be repeated back to the last parenthesis. If there is only a closing parenthesis, the phrase will be repeated back to the beginning of the string. As an example:

```
10 PLAY "abCDEFG)"
```

will repeat all seven notes. Double closing parentheses:

```
10 PLAY "O2CEGA))"
```

will cause an *infinite* repeat. This is particularly useful for things like repetitive bass lines. To turn off an *infinite* repeat you will need to use the H command.

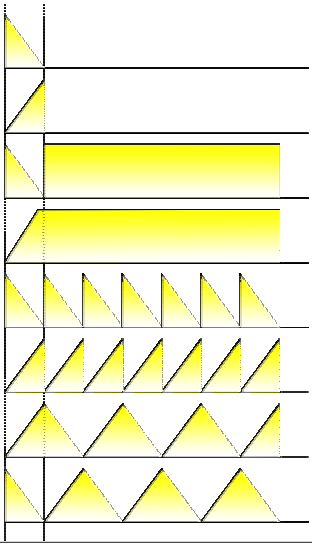
| Effect Value | Visual Representation | Description |
|---|---|---|
| 0 |  | Decay then stop |
| 1 | | Attack then stop |
| 2 | | Decay then hold |
| 3 | | Attack then hold |
| 4 | | Repeated Decay |
| 5 | | Repeated Attack |
| 6 | | Repeated Attack-Decay |
| 7 | | Repeated Decay-Attack |

*Table 12 – Volume effects values*

## The H command

An **H** included in any string immediately turns off the **PLAY** command. The main use of this is where you have an infinitely repeated bass line in one string. You can stop this at the end of the tune by putting an **H** on the end of the string which plays the melody.

## Comments

You can include reminders and comments anywhere you like by using **!!** marks. Anything written after a **!** will be ignored until the next **!** or the " at the end of the string is reached, for example:

```
10 PLAY "abCDEFG!chorus!aCEaDG"
```

## Channel selection

The command **M** is used to select which of the three *channels* are in operation per *PSG* and whether these give *noise* or *musical tones*.

You can have a maximum of *nine channels* (*three* per *PSG*) in use at any one time, but it does not matter whether they are all *tone*, all *noise*, or a mixture of both.

Your choice is entered with a number following the **M**, worked out like this:

| | Tone Channels | | | Noise Channels | | |
|---|---|---|---|---|---|---|
| Channel | A | B | C | A | B | C |
| Number | 1 | 2 | 4 | 8 | 16 | 32 |

*Table 13 – Channel audio type selection codes*

Mark each *channel* you want to turn on, and note down its number from the table above. Then just add them together to get the code you should use after the **M**. For example, if you want to use *tone channels* **A**, **B**, and **C**, you add the numbers **1**+**2** +**4** = **7**, so you use the command **M7**. In the same way, **M56** would turn on *noise channels* **A**, **B**, and **C**.

*Noise* can be used on any *channel* but the most wide-ranging frequencies are available in *channel* **A** for each *PSG*. For the best results, put your sound effects in the string which controls this channel for each *PSG* – *1st*, 4th and 7th string, in other words the first string per *PSG* after the **PLAY** command.

## Stereo control

The **PLAY** commands **L**, **R** and **S** control the stereo image for each *PSG*. The first two restrict the current *PSG*'s audio output to *Left* and *Right* speakers respectively while the latter resets the Stereo image. If your ZX Spectrum Next is set up with **ABC stereo** (the default), normally *channel* **A** goes to the left speaker, **B** goes to left and right, and **C** goes to right.

Therefore, if the **L** command is used, only *channels* **A** and **B** from the current *PSG* will be audible. Similarly, if **R** is used, only *channels* **B** and **C** will be audible. Like the **M** command, the **L**, **R** and **S** commands need to be re-entered in the strings targeting each *PSG*.

## Digital Audio

Your ZX Spectrum Next also contains hardware that can output digital audio, that is sound previously recorded digitally for reproduction, in a similar manner to your house or car CD and MP3 players. There is no easy way to manipulate this hardware from *NextBASIC* so *NextZXOS* provides several *dot commands*[1] (more on *dot commands* in *Chapter 19 –*

---

1 *Dot commands are short programs residing in folder* ***c:/dot/*** *which are used to extend* ***NextZXOS****, or to expose facilities not normally available to NextBASIC to the user. Dot commands were originally created for* ***esxDOS*** *(an alternative, free, ZX Spectrum–compatible Operating System which also works on the ZX Spectrum Next) and whose format was adopted by* ***NextZXOS*** *via its* ***esxDOS*** *emulation layer. Most* ***esxDOS*** *dot commands will work with* ***NextZXOS*** *and vice-versa unless they use some special facility not covered by either the* ***esxDOS*** *emulation layer or they are OS or machine dependent.*

*NextZXOS and alternatives*), written by David Saphier and Kev Brady, that can be incorporated into your programs and which not only allow you to play any **WAV** file stored on *SD Card* media but also a plethora of digital audio formats.

Currently via *NextZXOS* you can play natively (you'll see why we explicitly mention it in a second) the following audio file types:

### WAV

This is the standard audio format for most computers. The ZX Spectrum Next supports audio resolutions up to 32KHz. In order to playback a digital audio wave file, type:

```
.wavplay32 file.wav
```

where **file.wav** is the audio file you want to play. This can be accessed (like all other *NextZXOS dot commands*) from the 48K BASIC environment as well and fully incorporated into all your *NextBASIC* programs. You can find more information on how to access the digital audio hardware of your ZX Spectrum Next in *Chapter 22 – IN, OUT and the Next Registers .*

### MOD

MODule files are one of the major standards for computer music and they comes from the Amiga and its ProTracker application. There are two ways you can play MOD files on the Next. You either use the dot command **.nxmod** with your selection of **.mod** file as an argument; for example:

```
.nxmod song2.mod
```

or you can use the native application **NXModPlayer**. This can be found under **c:/apps/audio/NxModPlay/** accessible either via the Browser (See relevant section on the Browser in the next Chapter) or via the commands:

```
a$ =
        "c:/apps/audio/NxMod/nxmod
        play.nex"
.nexload a$
```

### PT3

PT3 is one of the de-facto standards for AY chip music, and the ZX Spectrum Next supports playback of up to 6 channel audio in two ways. First is via the dot command **.playpt3** with the pt3 filename as an argument:

```
.playpt3 onlyyou.pt3
```

or via the application NextSID. This is a rather special application as it not only allows you to play back pt3 music files but also to apply SID-like effects to the channels. NextSID can be found under **c:/apps/audio/NextSID**. As with NXModPlayer above you can either start it via the Browser or via the commands:

```
a$="c:/apps/audio/NextSID/nextsid.nex"
.nexload a$
```

Note that you will need a mouse installed.

### Using the Pi accelerator for audio

If you have the *Accelerated* version of the ZX Spectrum Next, or have a *Raspberry Pi Zero* installed on your board, then you have more options available audio-wise. These include (but are not limited to) playback of:

- Commodore 64 SID files
- “Tracker” MOD files

- Atari ST SDH files
- MP3 files
- High definition wav files

and many, many more.

The way the system works is as follows: The ZX Spectrum Next communicates with the Accelerator via its secondary *UART*[2] and sends commands and audio files to the specialised *SUPervisor* software that is running on the *Raspberry Pi Zero*. The *Pi Zero* in turn interprets these files and reproduces the audio contained therein via it's GPIO port onto the ZX Spectrum Next *I²S*[3] port which in turn mixes it with the rest of its audio output and redirects it to whichever output you have available. In essence when it comes to playback, the ZX Spectrum Next is considered a "sound card" where the accelerator is concerned and two extra DACs where the ZX Spectrum is concerned. As a consequence you can have Digital Audio (on the ZX Spectrum Next), all three PSGs playing AND Digital Audio (on the Pi Zero) all playing simultaneously!

To use the Pi audio facilities you need to first enable the secondary UART and set it to the accelerator. In *NextBASIC* or the *Command Line* you must type:

```
CD "c:/apps/rpi"
```

and press ENTER. Then type:

```
LOAD "pi.bas"
```

You'll get a message stating 9 STOP statement, 50:1 indicating the system is now ready to play audio using the Pi Zero. Feel free to poke about the listing of the PI.BAS program as it shows you the usage of *Next Registers* (see *Chapter 22* for more*).*

Playing audio files requires a dot command called .pisend which you can find in c:/dot/ which serves a two-fold purpose: to send files to the *Pi Zero*'s temporary storage *and* send the appropriate command for it to play. Thankfully D. Rimron-Soutter and David Saphier, maintainers of NextPi2[4] and .pisend respectively, have packaged all this nicely into little *NextBASIC* programs (located in c:/nextzxos/) which you can either call directly or via the *Browser* by selecting a *filetype* already registered. Currently registered *filetypes* include .SID, .MOD, .XM, .TZX and .SDH.

To illustrate how this works, we shall attempt to play an Atari™ SDH file. Assuming you have a SDH file named warhawk.sdh (search for it and download it on the internet; it's freely available) on the root of your SD card, playing it is as simple as:

```
LOAD "c:/nextzxos/sndplay.bas":
f$="c:/warhawk.sdh":GO TO 10
```

The screen will read Playing... c:/warhawk.shd and the music will start playing from your speakers.

## External Audio Output

If you are interested in doing more with sound from the ZX Spectrum Next, like hearing the sound that BEEP and PLAY make on something other than the usually limited audio of your display, you will find that the audio signal is also present on the *Audio Out* socket on the back of the machine. You may use this to connect to a pair of headphones or a higher quality amplifier. Note that this will not disrupt audio reproduction on the digital display cable, therefore you may want to turn down the volume on your display before plugging an

2 *UART or Universal Asynchronous Receiver-Transmitter is a hardware device that exchanges data sequentially between two systems. In our case this is done between the ZX Spectrum Next hardware and the Pi Zero accelerator via its GPIO port.*

3 *I²S or Inter-IC Sound is a serial bus interface standard to connect digital audio devices.*

4 *NextPi/2 is the operating system running on the Pi Zero accelerator that's purposely built to support the Next.*

external audio reproduction device. Note also, that there is no volume control for the *Audio Out* socket so you should take that into account when using headphones or an amplifier.

**Notes**

TZX files are "perfect" ZX Spectrum tape images. Due to them being compressed, they require a much more powerful CPU than the Z80N present on the Spectrum Next in order to be decompressed to their original tape audio stream. While not audio in the strict sense we're discussing in this chapter, they do use the audio subsystem to be loaded on the ZX Spectrum Next side and as such they are covered here.

## Exercises:

1. Rewrite the Mahler program so that it uses FOR loops to repeat the bars.
2. Program the computer so that it plays not only the funeral march, but also the rest of Mahler's first symphony.
3. Repeat exercises 1 and 2 above by utilising PLAY instead of BEEP.

# Chapter 19 – NextZXOS and alternatives

## Guide to NextZXOS

Until now, we have been talking about *NextBASIC*, the programming language with which you "talk" to your ZX Spectrum Next and get it to do things. Underneath *NextBASIC* however, lurks another program, one that allows your computer to communicate with the hardware devices connected to it and the world at large. It manages your computer's memory, makes sure your data is safe and accurate, that your programs behave as intended by their programmers and performs important "housekeeping" on your storage devices. This program is called an *operating system* and in the ZX Spectrum Next's case it is called *NextZXOS*.
*NextZXOS,* written by Garry Lancaster, is the direct successor to his *+3e/IDEDOS,* which in turn comes directly from the first proper Sinclair ZX Spectrum *operating system* called *+3DOS* which first appeared on the ZX Spectrum +3.

## NextZXOS main features

*NextZXOS* extends *+3DOS, +3e* and *IDEDOS* and features the following:

- *FAT16* and *FAT32* support for industry-standard compatibility with mass storage devices while retaining *IDEDOS*/*+3DOS* compatibility for a full range of storage choices
- Long File Name (LFN)[1] support
- Proper subfolders/subdirectories
- Memory Management facilities
- Virtual (container) *file systems* in *disk* and *tape images*[2]
- Installable device drivers
- Menu-driven file manager with extensible filetype associations/launchers
- *esxDOS* emulation layer for interoperability across ZX Spectrum compatible machines and extended *dot command* support
- Automatic execution of software on boot
- Command-line interface
- *Streaming* support
- Virtual memory support (*swap partitions*)
- Timekeeping facilities
- Availability of disk and file management even on legacy (via *dot commands*), 48K modes
- Increased compatibility with previous models of ZX family of computers[3]
- Support for a variety of *snapshot* formats
- Multi-lingual and multi-font capabilities
- Extended windowing facilities
- Increased speed of operation compared to the previous versions
- Proper CP/M[4] 3 compatibility

Unlike other operating systems, *NextZXOS* tightly integrates with the in-built programming language *NextBASIC*, to the point that it can be mistaken as being part of it. In reality however, *NextZXOS* provides two rich *APIs* (one being the native *NextZXOS API* and the other

1 *Long File Name support means that a filename under NextZXOS can be up to 255 characters long as opposed to the earlier 11 (8 for filenames +3 for extension/filetype) character limit . LFN capability is not reserved for files. Folders can also be up to 255 characters long. Longer file and folder names help with the organisation of your files as it is easier to use more descriptive names.*

2 *A container file system / disk image is a bit-for-bit copy of the contents of a mass storage medium contained within a single file. For example what used to be an entire floppy disk can be represented by one file, which NextZXOS will access with traditional disk and file management commands once this is attached (mounted) by the operating system.*

3 *NextZXOS is compatible –via emulators provided by Paul Farrow– with ZX80 and ZX81 while also being more compatible than its predecessor with Timex Sinclair as well as the 128K and 48K lines of ZX Spectrum machines.*

4 *CP/M is an older operating system for personal computers with a vast library of software.*

the *esxDOS-compatible API*) which can be used from *machine-code* or a language other than *NextBASIC* (for example *C*) to provide them with all the facilities needed for accessing your ZX Spectrum Next without having to write *low-level* access to the computer's hardware from scratch. The distinction is subtle and more easily discernible in facilities that are exposed to the 48K legacy mode of operation where the *NextBASIC* commands do not exist and their place is taken by the aforementioned *dot commands*.

In the following sections we will examine the *NextBASIC* usage of *NextZXOS* facilities before we extend the discussion to *dot commands* and the *NextZXOS Command line so for the next few sections you can approach the subject as a NextBASIC* topic if you feel more comfortable that way.

Let's however start by introducing topics in the order they will be needed in our discussion.

**Notes**

In this chapter, a lot of commands produce visual feedback that may be easier to see and understand on a 64 or 85 column display. Although *NextZXOS* menus are covered much later in the chapter, it may be of benefit to learn to use the *Command Line* in combination with the *32/64/85* option. You get to the *Command Line menu* either from the main *NextZXOS menu* or by pressing **EDIT** and navigating to it while in *NextBASIC*. Pressing **EDIT** again will allow you to select the *32/64/85* option which will cycle through all available widths until you find one that visually satisfies you.

## Files, Drives, Partitions and Disks

Like most Operating Systems, *NextZXOS* uses the concept of *Files* to store data in a hierarchical organised set called a *Drive* identifiable by a *Drive Name.* This is the combination of a letter[5] from **A** to **P** suffixed by a colon ie. **d:**, which in turn can be contained within a *Disk*. A *file* is any type of collection of data; *Sprites, Arrays, NextBASIC programs, machine code, images* or collections of the above. While *NextZXOS* via *NextBASIC* supports a finite set of file types, this set can be extended with the use of external programs and *dot commands*. For the following sections we will concentrate to what is available via *NextBASIC* and the provided *dot commands* with a brief discussion of how *NextZXOS* (and *NextBASIC* in turn) can be extended to handle more file types.

*Files* are usually organised in *folders*. While *folders* are not necessary for the storage of *files*, they are advisable as they help categorise and group *files* in a logical way, which allows them to be searched and accessed easily. That becomes apparent as your collection of *files* grows from a few tens to hundreds or thousands.

As mentioned above, *files* themselves are stored on *disks*, which are the physical *devices* that can be removed from the computer and whose contents are not lost like the main memory after each power cycle. Depending on the type of *disk*, there may be one or more data structures on it called *partitions* which as the name implies is a way to virtually organise the available space on the *disk* into smaller units. *Partitions* can be assigned to *drives* or sit unused –with or without data– invisible to *NextZXOS* (until a *drive* is assigned to them).

Apart from the *physical disks*, *NextZXOS* also allows the use of *virtual disks* and *tape images*. These are special files that contain an exact replica of the medium they simulate. They too, can be assigned to drives (see the footnote regarding tape images) as physical disks can and they appear to the user (and *NextBASIC*) as any other physical disk. There are some special considerations regarding these special files which we will visit further in this chapter.

## Working with files

In our examples in the previous chapters we have already used *files* and specifically one particular type of file: *NextBASIC* programs. Even more specifically, we have **SAVE**d and **LOAD**ed them by using two commands: **SAVE** and **LOAD**.

5 *NextZXOS cannot assign all letters in the range A to P as drives, since some are reserved;* ***C:*** *is always the boot drive,* ***M:*** *is the RAMdisk and* ***T: (an exception to the A to P range)*** *is the tape.*

Apart from that basic functionality; we can also *copy* or *move* files from one location (folder or drive or a combination of both) to another location, *rename* them, erase them, and *catalogue* them; that is to produce a list of all the available files in a location. These functions are possible with the use of the **COPY**, **MOVE**, **ERASE** and **CAT** commands or their *dot command* equivalents: **.cp**, **.mv, .rm** and **.ls**[6].

## Filenames

Before we visit the commands that manipulate *files*, it's best we visit the subject of *filenames* first as there are special considerations on how and why a *file* is named.

First of all, *filenames* are basically strings that are made by up to four parts (according to which *file system* we use as we will see further below) that help *NextZXOS t*o uniquely identify a file. These are:

- *User Area* with *Drive Name* –or–
  *User Area* followed by a colon (**:**) character if accessing files on the same drive –or– *Drive Name*
- *folder* name or combination of *folder* names separated by forward (/) or backward (\) slash characters
- *actual file* name
- suffix of a dot (**.**) character followed by a *file type* of up to three characters (for example **.bas**)

Of these only the third part is absolutely required and every other part is optional. Also not every part is applicable everywhere in *NextZXOS*. This strictly depends on the kind of *filesystem* the *files* are located on. For example you can only use *User Areas* on *virtual disk images*, the *RAMdisk* and *IDEDOS (+3e) partitions* but not on *FAT partitions* (we will examine these a little later), while you cannot use folders in the *RAMdisk* and *virtual disk images* as the concept of *folders* doesn't exist there[7]. Similar for *tape images* or actual tapes where you can only use *drive names* (specifically **t:**) and up to 10 charactes as a *filename* but not *folders*.

*Filenames* can be up to 255 characters in length (inclusive of dot character and the optional *type), however, for compatibility reasons on virtual disks*, *IDEDOS partitions* and the *RAMdisk*, they can only be 8 (*name*) + 3 (*type) characters in length (excluding the optional user area* and *drive* letter combinations).

Finally, some characters are reserved and cannot be used to name files. Files can use the following characters:

- Letters: **abcdefghijklmnopqrstuvwxyz** (upper or lower case)
- Digits: **0123456789**
- Other characters[8]: # $ @ ↑ _ { } ~ £

Upper and lower case letters are considered as having the same value for *filenames*, so *EXAMPLE* and *example* would be identical as far as *NextBASIC* is concerned. They will however be listed in the case they were stored in, when a *catalogue* is requested.

A *filename* can end with an optional *type field* which is just up to three characters[9] long that you may wish to use in order to group together or quickly identify files of the same type. If a

---

6 *SAVE and LOAD do not have dot command equivalents as they're already available in the 48K mode personality even though the latter was conceived prior to the introduction of mass storage devices to the ZX Spectrum family of computers.*

7 *Technically for floppy disk , IDEDOS and Tape virtual images as well as the RAMdisk slash characters can be a part of a filename but they're not an organisational unit as the folder is and since (as we'll see later) the filenames in these cases are restricted in size, it's not advisable to use them.*

8 *Characters " and ' are available in some situations (for example for tape images or for CP/M) for filenames but are reserved under NextBASIC and cannot be used directly.*

9 *Type fields, separated by a dot from the name field, are up to 3 letters long as a matter of both compatibility and convention. In reality, in FAT drives like the System/Next™ card your ZX Spectrum Next came with, there are no restraints on how many dots a filename can have but any filename with the dot character located at more than 4 characters before its end, is considered to have an empty type field (always keeping within the maximum allowed length of a filename). See also the discussion regarding wildcards to see why this useful to know.*

*type field* is specified, it must be preceded by a dot. Unlike some other BASICs, *NextBASIC* does not automatically allocate a *type* to files if one is not specified.

You may find it useful to add your own *types* – a popular convention is to use .BAS to identify *NextBASIC* file *types* and .BIN or .COD to identify machine code file *types*.

*NextBASIC* already understands a number of popular *types*. Going to c:/nextzxos with the browser, placing the cursor over browser.cfg and pressing ENTER will return the most commonly used ones together with the action that will be taken when the *Browser* launches them.

The characters * and ? are called *wildcards* and have a special meaning to *NextZXOS*. They're used to substitute ranges of characters or specific characters in *filenames* and *folders*. We'll see why this is particularly useful further below.

The dot character . also has a special meaning according to how many we use. If we use one (.) it means *this folder* and if we use two (..) it means *the folder one level up*. Keep this information in mind as it will prove very useful in the examples we'll encounter.

The following are some examples of valid *filenames*:

- z
- squares
- m:picture.bin
- a:fred
- 13a:hello
- 0M:CAPITALS
- file name
- test.bas
- philip
- glass.mus
- a:a.a
- c:/nextzxos/browser.cfg
- c:\nextzxos\browser.cfg
- 7:dubious

while the *filenames* below are illegal and attempting to use them will produce an error:

- <>-+=!& (must not contain any of these characters)
- *test (cannot contain an asterisk)
- te?st (cannot contain a question mark)

Note that in the list above we've made two assumptions regarding valid *filenames*, and these are that *drive names* a: and m: are *virtual disks* and the *RAMdisk* respectively. *User areas* are acceptable parts of *filenames* ONLY if the *drive*'s *filesystem* allows them; otherwise you will get an error.

With that information in hand, let's start examining below the main commands for working with files.

## LOAD

LOAD as its name implies retrieves a *file* from a *drive* and puts it (*loads* it) in the computer's memory. Depending on how it was saved (in the case of *NextBASIC* programs) or named (in the case of machine code software) it may also execute it as well. It takes the form:

LOAD *filespec* [*MODIFIER [options]]*

where *filespec* is a *filename* as described in the previous section followed by an optional *MODIFIER* directive (SCREEN$, LAYER, CODE, DATA, INT, INPUT or BANK) which in turn may have optional parameters.

Regarding the *filespec*, this can be as simple as an empty string, however this has special meaning for tapes and disk images. Typing:

```
LOAD ""
```

will produce an **F Invalid file name, 0:1** error. We'll revisit this promptly but first let's type:

```
LOAD "t:"
```

If you now repeat the previous command, you will see something changing on your screen, with its border turning red and the rest of the screen becoming blank. This simply means that your ZX Spectrum Next is expecting a tape to load! Indeed, finding a tape deck, connecting it to your computer and a ZX Spectrum program on tape, inserting it and pressing PLAY you will start seeing blue and yellow bars running down the border and the program eventually loading. What the series of commands we just typed did, is to first switch the *default LOAD device* to tape (that's denoted by the drive name ***T:***) as opposed to the SD Card and then attempted to load the first program on the tape that it could find. Pressing ***SPACE*** or ***BREAK*** will return you to *NextBASIC* without loading anything. There is a shortcut of the previous series of commands in the form of the *Tape Loader* option in the *NextZXOS Startup Menu*. This is also the preferred way of loading tape-based software on your ZX Spectrum Next. Using **LOAD** with only a *drive name* as parameter will set the *default drive* to that drive and all file operations not having a *drive name* specified in the *filespec* will assume it.

We already learned that *filespec* can be only a *drive name*. There is one more special case and this concerns *virtual disks*. Obviously, unlike what happens with a tape, the concept of *the first program you can find* cannot exist on a random access medium like a disk, so **LOAD ""** will produce the error we saw when we first attempted it. In virtu*al disks how*ever it is possible to give the command:

```
LOAD "*"
```

This will attempt to load a special file named *****, or, in the absence of that, load a file called **DISK**. As we saw earlier, you cannot use *NextBASIC* to name a file * as this character is a *wildcard*; you can however save a file called **DISK** and this will be loaded and if saved with the appropriate **SAVE** option, will also execute. You can try this by pointing the Browser to **c:/demos/NextBASIC/** and selecting **demo.dsk** as a *virtual disk*, when prompted to mount it, select **A** and then **N** (when asked if you would like to *Autoboot* it). Then just type the command above and you'll be greeted by a cheerful **Hello World** message.

A bit earlier, we discussed how wildcard characters can be useful. We saw how it is to use one as *filespec* in **LOAD** which as we said is reserved only for virtual disks. A variation to that which uses the ***** *wildcard* is the following:

```
LOAD "d*"
```

which will attempt to load the first *NextBASIC* file that starts with the letter **d**. We'll revisit *wildcards* further below as they're a very powerful tool for manipulating files.

So far we've examined **LOAD** with only the *filespec* option. This will load *NextBASIC* programs into memory, however with the optional use of *MODIFIER* directives, **LOAD** can display pictures, retrieve long data segments and load either code or raw data into memory.

One of the nice facilities provided by *NextBASIC* is the ability to store the screen as it's being displayed at a given moment, in order to be loaded later and redisplayed instantly, whether it contains graphics, text or both. There are two (plus one) ways that this can be achieved; first is with the use of **SCREEN$** and second is with the use of the **LAYER** *modifiers*. Here we'll skip ahead as we haven't talked about **SAVE** yet but for the time being type the following:

```
10 LAYER 0
```

```
20 INK 3: PAPER 6: PRINT
   "Hello World!"
30 SAVE "test.scr" SCREEN$
```

and then RUN it. You will immediately be greeted by purple letters on yellow background.

Now type:

```
CLS:LOAD "test.scr" SCREEN$
```

Immediately, the same message as previously will appear on your screen. Now change line 30 and replace SCREEN$ with LAYER so it reads:

```
30 SAVE "test.scr" LAYER
```

and RUN it again. Then give the following:

```
CLS: LOAD "test.scr" SCREEN$:PRINT AT
2,2; "Press Any Key": PAUSE 0: CLS: LOAD
"test.scr" LAYER
```

and press ENTER. What you will see is two consecutive LOADs of the same image with an intermediate prompt to press a key. Before we explain what just happened, type one more thing:

```
LAYER 2,1: LOAD "test.scr" SCREEN$: PAUSE
0: LAYER 2,0: LAYER 0
```

This will produce a blank screen waiting for a keypress which when it comes will give its place to the screen you previously saved. Finally modify the above line slightly to be:

```
LAYER 2,1: LOAD "test.scr" LAYER: PAUSE
0: LAYER 2,0: LAYER 0
```

which will produce a garbled image and an error report End of file which will disappear once you press a key. Don't forget to manually turn off *Layer 2* after that command because due to the error, the command did not fully execute. What has happened is that the LAYER modifier attempted to load a screen in the format supported by the current layer as set by the LAYER command (and then run out of data as the *Layer 0* screen we saved is markedly smaller), while SCREEN$ exclusively loads screens in the format recognizable by *Layer 0*; That means that for *Layer 0*, LOAD … SCREEN$ is functionally equivalent to LOAD … LAYER but that doesn't apply to the other layers. LOAD *filespec* SCREEN$ and LOAD *filespec* LAYER *do not* store the current palette in use. If you haven't changed the palette at all and are using *NextBASIC*'s standard colours, then you'll get the display you're expecting, however if you have changed the palette you may be surprised by the unintended effects this can produce. In order to get the active palette and store it in a file you will need to use the ZX Spectrum Next's *NextREG* facilities covered in *Chapter 22 – IN, OUT and the Next Registers,* or the very handy *Save Palette* function of the *NMI menu* covered later in this chapter. Additionally you cannot load screens in the shadow areas of the graphic subsystem. For that you will need the following LOAD *modifier*; CODE with optional parameters *address*, *length*. This essentially loads *machine code* programs and *raw data* into memory either in the *address* they were saved from, or in the *address* and *length* –in bytes– we specify. Keeping with the example above, type the following:

```
LAYER 0: LOAD "test.scr" CODE
```

Once again, you'll be greeted by the cheerful Hello World! screen we generated previously. To expand a bit on this first type:

```
NEW
```

After pressing ENTER, you'll be greeted by the *NextZXOS Startup menu*. Select *NextBASIC* and rewrite the line above by adding 16384,6144 at the end after the CODE to read:

```
LAYER 0: LOAD "test.scr" CODE 16384,6144
```

Amazingly, the Hello World! message reappears but this time colourless! Adding the two numbers after CODE instructed the computer to load the file in address 16384 (which is the start of *Layer 0*'s graphics memory) but at a smaller length than the actual file we've stored, removing all the colour attribute information. Attempting to set a longer length than the size of the file we're loading, the computer will return an End of file, 0:1 message. Note here that doing just that is not a good practice and we should be using LOAD BANK, LOAD SCREEN$ and LOAD LAYER to load data into graphic memory.

As we saw in *Chapter 11*, one of the most tedious aspects of programming is to prepare arrays. They can involve endless typing via data statements and use a lot of program space which could otherwise be used for actual program logic. Thankfully *NextBASIC* gives us the option, after we've prepared an array, to save it to a file to be retrieved later, saving us both time and code memory. To load such prepared arrays we need to use the LOAD *modifier* DATA. This takes the form:

LOAD *filespec* DATA *arrayname()*

to load for example the array b() from *Chapter 11* assuming we have already saved it as b-array.dat we'd only need to type:

```
LOAD "b-array.dat" DATA b()
```

This would find if any other array named b() was already stored in the computer's memory, erase it and replace it with the information provided in the file.

We can only load string and floating point arrays. Also of note is that the parentheses after the array name cannot be ommited.

Integer arrays cannot be loaded or saved with the DATA modifier. For that we need the following modifier INT.

LOAD *filespec* INT

will load previously saved integer arrays and variables. All arrays and integer variables will be initialised prior to loading and replaced with what is in the saved file. Additionally the INPUT modifier:

LOAD *filespec* INPUT

will load a previously saved definition of the keyboard joystick. See *Chapter 14* and the INPUT function for more.

The final LOAD *modifier* BANK should be looked upon as a variant of the CODE modifier, as it basically loads raw data into memory in the *bank number*, *offset* in said bank and *length* (in bytes) we specify very much like CODE does. This takes the form:

LOAD *filespec* BANK *number*, *[offset]*, *[length]*

Keeping with the example we have been using try:

```
LOAD "test.scr" BANK 5
```

will load and display the exact same screen, with the main difference that it will put it in *offset 0 of bank* 5. For reasons that will become clear in *Chapter 23*, this is exactly the same location as the one we used with SCREEN$ and therefore if you slightly modify the command to be:

```
LOAD "test.scr" BANK 5, 0, 6144
```

as previously, the file will appear colourless. When using BANK as a LOAD *modifier,* we need to remember that *NextBASIC* and *NextZXOS* do not care what type of data is being loaded. As such the BANK *modifier* is also used to load *NextBASIC* programs that make the use of banks. More about that below when we examine SAVE.

## SAVE

Our computer's memory lacks permanence; whatever is stored inside it during operation disappears when we turn the power off. We need some means to store the information onto a medium that can hold it even when the power is off; this comes in the form of the SAVE keyword.

It follows the exact syntax of LOAD that we examined in the previous *section* and uses the same *modifiers* and *parameters* with an additional LINE *modifier*. There are a few differences from LOAD in behaviour however and we'll examine these immediately. Typing:

```
SAVE ""
```

will produce an F Invalid file name, 0:1 error even when our *default drive* is T: (tape). That's simply because even on a tape, files NEED to be named, otherwise we wouldn't be able to identify them!

As with LOAD, setting the *filespec* to a *drive name* (for example c:) will switch all *NextBASIC* file retrieval and storage operations to that drive from that point forward so for example:

```
SAVE "m:"
```

will make *drive* m: the default *drive* and won't actually store any information anywhere.

As we saw in examples in the previous section, SAVE *filespec* without a modifier (assuming *filespec* is a string specifying more than just a *drive name*) will save the *NextBASIC* program currently in memory onto the *default drive* or the *drive/folder* we specify. If however this *filename* already exists in the location specified, *NextZXOS w*ill first create a *backup file* made up from the original *filename* and then append the *type* .bak to it.

We will have to skip ahead again to see the results of our operations by using CAT (for CATalogue) so let's quickly do some typing:

```
SAVE "c:"

10  PRINT "Hello"
```

and then:

```
SAVE "hello.bas"
```

followed by

```
CAT "hello*.*"
```

(Never mind what the *.* means, we'll examine that later).
Your screen will display the following:

```
hello.bas                          1K

  980M free
```

Now perform the save again, again followed by CAT "hello*.*" and you'll see:

```
hello.bas                          1K
hello.bas.bak                      1K
```

```
      980M free
```

before we discuss what has happened, make a small modification to the program (for example add an exclamation mark after World on line 10 and do another save, a bit different this time:

```
SAVE "hello"
```

and follow it by CAT "hello*.*" . Now you'll see:

```
hello                                1K
hello.bas                            1K
hello.bas.bak                        1K

  980M free
```

Repeat the last save command one more time and then do CAT "hello*.*" again. The screen now shows:

```
hello                                1K
hello.bak                            1K
hello.bas                            1K
hello.bas.bak                        1K

  980M free
```

If you however, had started with a SAVE "m:" thus redirecting the *default drive* to the *RAMdisk*, everything would have been a bit different. First by not displaying a hello.bas.bak and now after the entire series of commands CAT would have returned:

```
HELLO                                1K
HELLO.BAK                            1K
HELLO.BAS                            1K

      59K free
```

so, why the difference? Let's take it from the beginning. We initially saved a *NextBASIC* program that was named hello.bas; then once we saved it again, the file with the same name on the drive had a .bak *type* appended to it. Then we saved the same program with a name without a *type*. In the second case since we were trying to save to a *+3DOS filesystem* (the *RAMdisk)*, *NextZXOS* can only use 8+3 character *filenames* unlike the *FAT filesystem* that can have very long *filenames*. So in the second case, instead of appending the .bak *type* to the original hello.bas file, it stripped the .bas *type* and replaced it with .bak. What followed is, that we tried to save the same name without *type* but now *NextZXOS* had a decision to make; which *filename* with .bak *type* to keep? As you could easily find out by LOADing back the hello.bak file, the last version saved is the one retained. Your PRINT statement would be the one with the exclamation mark and not the one without.

This example, makes an important point that due to the disparate types of *filesystems NextZXOS* can handle, the *auto backup* feature provided is nice but it's not a panacea, so do not rely on it exclusively and instead name your files explicitly!

A slight variation of the SAVE command as it deals with *NextBASIC* programs is that you can add the LINE *modifier* with either a numerical parameter or a label name after it. For example saving the program above with:

```
SAVE "hello.bas" LINE 10
```

and then doing

```
LOAD "hello.bas"
```

will load AND start the program at line **10** or at the specified label[10] which will then print **Hello** on your screen. As a matter of fact you can use even non-existing line numbers when saving. **LOAD** will go to the first available line after the one you entered if that doesn't exist in your program and attempt to run from there. If the line number you entered is higher than the last line number in your program, **LOAD** will just not execute the program, just simply loading it as if the **LINE** *modifier* was never specified. **SAVE** *filespec* **LINE** *number* will NOT accept a number greater than **65535** however and it will return a **B Integer out of range, 0:1** error if such a value is supplied for *number* or **0:1 No Label** error if the label doesn't exist.

It is noteworthy, that a particular *type* is not forced upon the file when using **SAVE**, so a *NextBASIC* file for example will not automatically carry the *type* **.bas**. That being said, as we saw earlier a standard set of *types* is known to the *NextZXOS browser.* These, help it automatically launch files using the appropriate commands. It is therefore a good idea to either adopt these, or modify the ones known to *NextZXOS t*o be the ones you prefer. Remember however that every time you update **System/Next™**, the known associations to *file types* are being overwritten with the default ones, so always keep a backup of the **browser.cfg** file located in **c:/nextzxos/** if you indeed make these changes.

As we saw earlier, storing screens requires the use of either the **SCREEN$** (for Layer 0) or the **LAYER** (for all other layers) *modifier* directives. From our examples, you may have already assumed that the **LAYER** *modifier* this can also be substituted by the **BANK** or **CODE** *modifiers*. While this is true for *Layers 0* and *1*, there's no functional way this can be done for *Layer 2* with **CODE** or **BANK** as the latter occupies more than one banks and **CODE** only works within the main memory map.

The most compatible way to save screens is therefore the use of the **LAYER** modifier directive as follows:

**LAYER** *desired_layer*
*<statements generating graphical content>*
**SAVE** *filename.ext* **LAYER**

Remember, that you must already be in the layer that you intend to save before initiating a **SAVE...LAYER** command. Also, as you can find from looking at **browser.cfg**, *NextZXOS a*lready recognises some *types* as belonging to a specific layer screen file. The table below lists them in order:

| Type/Extension | Layer |
|---|---|
| .SCR | ULA (Layer 0) |
| .SLR | LoRes (Layer 1,0) |
| .SHR | HiRes (Layer 1,1) |
| .SHC | HiColour (Layer 1,2) |
| .SL2 | Layer 2 |

*Table 14 – Automatically recognisable screen file types*

By this time and given the time we spent discussing the **CODE** *modifier,* you've probably figured out that it's not reserved for machine code programs and instead will save or load the raw data that's located in the memory address you specify whether this is graphics, machine code, a *NextBASIC* program, variables, *NextZXOS system variables* or just random numbers or even nothing (0s).

Unlike its **LOAD** equivalent, **SAVE … CODE** requires both parameters, that is a *legal* address and *valid* length. It takes the form:

**SAVE** *filespec* **CODE** *start_address*, *length*

10 *Labels can be anywhere in a line however SAVE filespec LINE @label will place the autostart pointer to the beginning of the line where the label is located and not at the location of the label.*

where *start_address* can be any number from 0 to 65535 and *length* any number from 1 to 65535 and the sum of these should not exceed 65536[11]. CODE as discussed works only in the main memory (or rather in the main memory map) and for the rest of the memory we should use the BANK modifier. The main difference is that BANK is only 16K in size thus accepting a maximum of 16384 as *offset*[12] and *length*. BANK can be used without an offset or length (but once an offset has been specified, the length parameter is required). Saving the contents of a *bank* takes the form:

SAVE *filespec* BANK *number, [offset, length]*

For *NextBASIC* programs that make the use of *memory banks* (as we'll see in *Chapter 23),* apart from the main program that can be saved with a simple SAVE command, you also need to save all the *banks* that contain parts of the program. It is therefore imperative to use SAVE...BANK on its own (without offset information) to make sure that all the *NextBASIC* parts are saved. As you will also see it's good practice to also assign *banks* when writing a *NextBASIC* program using variables so when you're loading them back you do not have to literally assign specific bank numbers as these can be reused by *NextZXOS* or a machine code program already in memory.

We already saw how we can use LOAD to load arrays into *NextBASIC* without having to enter complex DATA statements that have the potential of making our program hard to read. We SAVE arrays by using the DATA *modifier* followed by the array name (including parentheses) we wish to store for later usage. A few things we need to note are:

We cannot use a non-dimensioned array in our SAVE statement. For example if we do:

```
SAVE "data" DATA a()
```

we're more than likely to receive a 2 Variable not found, 0:1 error. Writing something like this:

```
DIM a(3): SAVE "data" DATA a()
```

however will save happily.

An already dimensioned array can be saved using a direct *NextBASIC* command or as part of a program but a saved array loaded using the command line or a direct *NextBASIC* command will NOT be available from your program unless it's loaded explicitly from it. Let's illustrate this point by writing the following little program:

```
10 DIM a(30)
20 FOR f=1 TO 30
30 LET a(f) = 30-f/f
40 NEXT f
50 SAVE "data" DATA a()
```

RUN the program and then type NEW to restart *NextBASIC*. Then type the following program:

```
10 FOR f=1 TO 30
20 PRINT a(f)
30 NEXT f
```

11 *In reality NextBASIC, in order to retain compatibility with earlier versions of Sinclair BASIC, allows all valid integer numbers as both address and length. If you however include a non-valid length, you cannot be certain of what you're actually storing so make sure you verify that the locations you're storing are inside the actual memory map.*

12 *Using the term offset is more accurate than start address for a bank as it can move location in the memory map. Locations within a bank always start at 0 and that's common on all banks.*

If you RUN the program you'll get a 2 Variable not found, 20:1 error, denoting that at line 20, *NextBASIC* has no idea what a means. Now without erasing the program give the following series of commands:

```
LOAD "data" DATA a():FOR d=1 TO 30: PRINT
a(d): NEXT d
```

You'll get the same series of numbers you stored with the previous program (before you typed NEW) on screen. If you however attempt to RUN the program you just typed the 2 Variable not found, 20:1 error will persist. In order to fix this, you will need to add the following line:

```
1 LOAD "data" DATA a()
```

which will produce the same effect as the direct command you gave earlier. You do not need to DIMension the array as LOAD will do that for you. It is also useful to note that it doesn't matter which array's data you saved since, when you load the same data back, you can assign it to any available array. So you could theoretically SAVE "data" DATA a() and LOAD "data" DATA b(). The only thing you need to remember is that the array type must match the data saved otherwise you will receive a b Wrong file type, 0:1 error.

Using the INT modifier with SAVE will store a snapshot of all your integer variables and arrays. All 24 integer variables and 24 integer arrays are saved regardless of if they contain data or not.

Finally the INPUT modifier used with SAVE stores your current keyboard joystick key assignments

### VERIFY

When storing data on tape, in order to make sure what the program or raw data that you've stored is accurate, *NextZXOS p*rovides *NextBASIC* with the VERIFY command. On media other than a tape, VERIFY has no effect unless it's used in conjunction with a *drive name* in which case it will act like its LOAD and SAVE counterparts switching the default drive to the one specified. In every case, if not used on tape (drive t:), VERIFY will return 0 OK 0:1. VERIFY follows the same syntax as SAVE except for the LINE modifier. Assuming you have a tape deck attached to your ZX Spectrum Next, and having the Hello World! program we typed a little earlier, save the program into tape by giving:

```
SAVE "t:": SAVE "hello.bas"
```

Now we will try to make sure that the program was saved to tape properly by doing the following:

1. Rewind the tape to just before the point at which you saved the program.
2. Type...
   ```
   VERIFY "hello.bas"
   ```
3. Play the tape. The border will alternate between red and cyan until *NextZXOS* finds the program that you specified, then you will see the same pattern as you did when you saved the program. During the pause between the blocks, the message Program: hello.bas will be displayed on the screen. (When *NextZXOS* is searching for something on tape, it displays the name of everything it comes across). If, after the pattern has appeared, you see the report 0 OK, then your program is safely stored on tape and you can skip onto the next section, Otherwise, something has gone wrong – take the following steps to find out what.

If the program name has not been displayed, then either the program was not saved properly in the first place, or it was but was not read back properly. You need to find out which of the two is true. To see if it was saved properly, rewind the tape to just before the point at which you saved the program, then play it back while listening to your audio output.

The red and cyan lead-in should produce a clear, steady high pitched note, while the blue and yellow information part gives a much harsher screech.

If you do not hear these noises, then the program was probably not saved. Check that you were not trying to save the program onto the plastic leader at the beginning of the tape. When you have checked this, try saving again.

If you can hear the sounds as described, then SAVE was probably alright and your problem is with reading back.

It could be that you mistyped the program name when you saved it (in which case when *NextZXOS* finds the program on the tape, it will display the mistyped name on the screen). On the other hand, perhaps you mistyped the program name when you verified it, in which case *NextZXOS* will ignore the correctly saved program and carry on looking for the wrong name, flashing red and cyan as it goes.

If there is a genuine mistake on the tape, then *NextZXOS* will display an R Tape loading error which means in this case that it failed to verify the program. Note, that a slight fault on the tape itself (which might be almost inaudible with music) can wreak havoc with a computer program. Try saving the program again, perhaps on a different part of the tape (or a different tape altogether).

## MERGE

Many programmers like to store parts of their programs or special subroutines they want to use again and again, thus building *libraries* of code. Normally a subroutine will be part of a larger program but what if it could be used anew on a different kind of program? Normally you would have to load the entire program into memory, edit out the parts you do not need and then proceed to write the rest of the new program only leaving the part that you want to reuse intact. Similarly, there may be someone that only wants a routine to be used once into their program (for example during initialisation) and then exchange that space for another routine that performs a completely different task. The answer to both these issues is the MERGE command. MERGE is used in the same way as LOAD with the difference that it doesn't clear what's in memory already and does not erase the program's variables and instead only replaces lines that already exist. To illustrate this point consider this little program:

```
10 PRINT "Part 1"
20 PRINT "Part 2"
30 PRINT "Part 3a"
50 PRINT "Part 5"
```

Now save the program by giving:

```
SAVE "part-a.bas"
```

and then give the command:

```
NEW
```

After you re-enter *NextBASIC* and type LIST you will see there's no program in memory. At that point type:

```
30 PRINT "Part 3"
40 PRINT "Part 4"
60 PRINT "Part 6"
```

Now save this program also by giving:

```
SAVE "part-b.bas"
```

Finally load the first program again by giving:

```
LOAD "part-a.bas"
```

and doing LIST. What you're going to see is the first program as you expected. You should now type:

```
MERGE "part-b.bas"
```

and then type LIST. Both programs have mixed (merged) together with line 30 being the newer one. If you had done the procedure somewhat inverted, that is part-a.bas was merged into part-b.bas then line 30 of part-a.bas would be the newest one and it would have overwritten line 30 of part-b.bas saying PRINT "Part 3a" instead of PRINT "Part 3".

Like LOAD when used on tape (drive t:), MERGE does not need a defined *filespec* accepting instead just an empty string ("") and will just merge the next available program. Another good use of MERGE is instead of LOAD for programs that have been saved with the LINE *modifier*. MERGE will just load the program without executing it thus allowing you to edit instead of trying to use BREAK to stop execution. MERGE will not work with CODE, SCREEN, LAYER or BANK *modifiers*. To partly simulate that functionality, there's a *dot command* called .extract which we will visit later on. Finally, MERGE does not work with arrays (DATA).

## Using NextZXOS

Thus far, we have examined the major commands we can use to get files into the computer's memory, as well as store the contents of the computer's memory into files but with the exception of a slight glimpse into rudimentary cataloguing of files on a drive, we do not actually know how to manage the files. The following sections will cover all the facilities provided for file and folder management by *NextZXOS,* together with their *dot command* equivalents (the latter work on both *NextZXOS* proper as well as 48K mode and some even work on *esxDOS* which we'll cover at the end of this chapter). We will also examine the remaining features of *NextZXOS* as the system itself does much more than simple *file* and *folder* management. Let's start by examining a few concepts that are necessary in order to get a better grasp of the commands that will follow and what these do.

## Wildcards

Earlier, we touched briefly on the subject of *wildcards*. We mentioned two characters * and ?. Their meaning is as follows:

| | |
|---|---|
| * | Any number of characters up to the end of the *Name* part of the *filename* if used prior to a dot within the *filespec* –and– any number of characters remaining up to the end of the *Type* part within the *filespec* if used after a dot in the *filespec* |
| ? | Any single character |

As a note to the above, it important to remember that the *type* part of a *filename* is recognised by *NextZXOS* as a *valid* one, only if it consists of up to 3 characters. If there are more than 3 characters it is considered to be a part of the name field and the *type* is therefore considered blank.

You *cannot* use more than two * within a *filespec* and each * *must always be the last character* in its respective field (*Name* or *Type*) in the *filespec*, otherwise a Bad Filename 0:1 error will be returned. Below are some examples of proper and improper usage of wildcards:

These will work:

| | |
|---|---|
| *.* | *Any filename* with *any type* |
| * | *Any filename* without a *type* |
| *.? | *Any filename* with any SINGLE LETTER *type* |

| | |
|---|---|
| ***.??a** | *Any filename* with *any type* that ends in the letter **a** |
| **a*.???** | *Any filename* starting with **a** with *any type* |
| **??a.?b?** | *Any* three letter *filename* ending with the letter **a** with a *type* having a **b** as second letter (for example **dba.dbf**) |

While these won't:

| | |
|---|---|
| ***d.*** | ***** not the last character in the *Name field* |
| ***.scr.*** | ***** not the last character in the *Name field* |
| ***.*d** | ***** not the last character in the *Type field* |

As it's apparent from the examples above, combinations of very few characters can represent a wide array of *filenames* which is exactly why *wildcards* are invaluable in managing our files.

## Filesystems

We've also talked about *filesystems*; more specifically about *FAT* and *IDEDOS*/*+3DOS* but not specifically about what these represent. In a few words, a *filesystem* is a specific way of organising information that's located on a *storage medium*. There are *filesystems* that are medium–specific (for example even though it doesn't have a specific name, the way files are stored onto tape is a *filesystem* in itself) and *filesystems* geared toward general use. *NextZXOS* supports 3 (or rather 4) *filesystems*: the ZX Spectrum native tape *filesystem*, *+3DOS* (that comes from the ZX Spectrum +3[13] principally geared towards floppy disks), and two variants of the *FAT filesystem*, *FAT16* and *FAT32* (their main difference where *NextZXOS* is concerned is capacity). *FAT* is the de-facto standard *filesystem* for most modern removable media (like the SD cards the ZX Spectrum Next uses). Each *filesystem* has its pros and cons which affects slightly the way *NextZXOS* operates. As we've already noted earlier not all features are available on every supported *filesystem*; this obviously affects some of the features we'll examine below.

*IDEDOS* (which comes from the +3e) is not a *filesystem* in itself but a scheme that allows multiple *+3DOS* "partitions" to occupy a single physical disk, in order to facilitate the use of large media like hard disks.

## Partitions

In the introductory notes and the *Filenames* section, we've mentioned the term *partitions* either by themselves or in conjunction with one of the *filesystems* mentioned above e.g. *a FAT partition*. This is a bit misleading and in reality it's an acceptable mashing of two terms: *XXX filesystem type* AND *partition – a partition formatted with the XXX filesystem*. In other words a *FAT partition* is *a partition formatted with the FAT filesystem* (could be either *FAT16* or *FAT32* – using *FAT* as a portmanteau term is acceptable use).

But what is a partition? Nothing more than an arbitrary slicing of available space on a storage medium, usually to make it more manageable. An SD card for example could have one or more *partitions* and not all of the same *filesystem*. Note here that *NextZXOS* will always start from the first *FAT partition* on the first SD card on the system. If you remember the initial discussion, *drives* can be assigned to *partitions*; this process of assigning a *partition* to a *drive* is called *mounting* and we will examine it right after we briefly examine *storage devices*.

## Storage devices and disks

For *NextZXOS* a *storage device* can be *physical* or *virtual.* We use the term *disk* for both but the former refers to an actual, tangible piece of hardware like the SD Card reader your ZX Spectrum Next is equipped with, while the latter is nothing but a file containing the image of a *filesystem*. *NextZXOS* uses a common set of controls to address and access both types of disks. *Physical disks* are generally –with the exception of tape– assigned a number per device (ie. the primary SD card reader and secondary SD card reader have differ-

13 *The +3DOS filesystem is identical to the CP/M one.*

ent numbers) and each *partition* on each *disk* (if a *partition* exists) is assigned a number in turn. *Virtual disks* on the other hand do not have device numbers as they don't physically exist however both require a *driver*; that is a small program that sits between the *disk* and *NextZXOS* and translates each device's individual characteristics into the common set of controls that *NextZXOS* understands. That alone however is not enough; *NextZXOS* needs to assign a *drive* to each *partition* on a *disk* (or in the cases of *virtual disks* and the *RAMdisk* to the *disk* itself). As it comes with your **System/Next™** distribution; *NextZXOS* knows three types of *physical disks*: *SD Cards*, the *RAMdisk* and *floppy disks* and two types of *virtual disks*: +3 *floppy disk images* and *IDEDOS hard disk images*. It also knows *virtual* and *physical tapes* both addressable via the reserved drive **t:**. *Physical disk* device numbers start at **0** and are assigned according to the table that follows:

| Device Number | Description |
|---|---|
| 0 | All IDEDOS partitions on the first SD drive |
| 1 | All IDEDOS partitions on the second SD drive |
| 2 | Reserved for First Floppy Disk drive |
| 3 | Reserved for Second Floppy Disk drive |
| 4 | RAMdisk |
| 5 | All FAT partitions on the first SD drive |
| 6 | All FAT partitions on the second SD drive |

*Table 15 – Device Number assignments*

On an unexpanded ZX Spectrum Next with an unmodified distribution of *NextZXOS*, the first used number is **4** which is the *RAMdisk* and the second is **5** as **System/Next™** comes on an SD card containing only a single *FAT partition*. As seen on the table above, device numbers **2** and **3** refer to floppy disk drives (not yet supported by *NextZXOS*).

## Mounting

In order for *NextZXOS* and *NextBASIC* to know how to access a *partition* or *disk* (be it *physical* or *virtual*) this *partition*/*disk* has to be *mounted*. That is the process where a *partition* on a *device* gets attached to a *drive*. If freshly installed, *NextZXOS* will automatically mount two drives; drives **c:** and **m:** the first being device **5** partition **1** (in other words the **System/Next™** distribution's SD card plugged into the first SD reader of the system) and the second one being device **4** (the *RAMdisk*). On an initialised *CP/M* distribution (as we'll see further below) one more drive will be mounted and that's drive **a:** (assigned to **cpm-a.p3d** located inside **c:/nextzxos/**).

Generally speaking, if there are more than one *FAT partitions* detected on the SD card(s), they will be automatically mapped to drives **c:** onwards on startup.

Finally, any files located inside the **c:/nextzxos/** directory, are mapped to the appropriate *drives* (if the *drive* in question has not already been mapped), if they are named as follows and are valid *+3DOS partition images*:

**DRV-A.P3D**
**DRV-B.P3D**
(…)
**DRV-P.P3D**
**CPM-A.P3D**
**CPM-B.P3D**
(…)
**CPM-P.P3D**

*Virtual images* named **DRV-x.P3D** (where **x** is a letter from **a** to **p**) have preference over *virtual images* named **CPM-x.P3D** so in the presence of both, the **DRV-x** variant will be mounted. Apart from the *auto-mounting* procedures described above; we can also manually *mount partitions* and *disks*. This will be covered a bit further below at its own section.

With all this information at hand, we can now proceed to examine *NextZXOS* facilities by task.

## Drive cataloguing

It's obvious that simply remembering a file's name and LOADing it, is not possible after the first few files, so we need a command that can help us see which files are stored on a drive. This command is **CAT** (from CATalogue) and its syntax is as follows:

**CAT** [-] [*#n*[,]] [*filespec*] **[EXP]**

where - is a switch instructing the file list produced to use the short (8+3) format, *#n* is a *NextZXOS stream* for the output of **CAT** to be redirected to, *filespec* follows the conventions described in the *filenames* section earlier and the *modifier* **EXP** produces an expanded listing with more information about the files being listed. All **CAT** parameters are optional and by itself **CAT** will produce a listing of the *default drive* which can be set in the same manner as with **LOAD**, **SAVE** etc. Try the following:

```
LOAD "m:"
CAT
```

You will receive the following on your screen

```
No files found
    62K free

0 OK, 0:1
```

Congratulations, you just listed the contents of the *RAMdisk*. Sadly it's empty! Now type:

```
LOAD "c:"
CAT
```

Your display now will look similar to this:

```
DEMOS                                <DIR>
DOCS                                 <DIR>
DOT                                  <DIR>
GAMES                                <DIR>
MACHINES                             <DIR>
NEXTZXOS                             <DIR>
RPI                                  <DIR>
SRC                                  <DIR>
SYS                                  <DIR>
TMP                                  <DIR>
TOOLS                                <DIR>
LICENSE.MD                             6K
README.MD                              2K
TBBLUE.FW                            168K
TBBLUE.TBU                           465K

 1887M free

0 OK, 0:1
```

which is a slightly modified listing of the contents of the *root folder*[14] of your **System/Next™** distribution. Now type:

```
CAT EXP
```

14 *In filesystems other than IDEDOS and +3DOS that use User Areas, files are organised in an inverted virtual tree of sorts, contained in folders like branches on a trunk of a tree which in turn contain smaller branches and so forth. The top level of the tree is called the root folder or root directory.*

Your display now will look similar to this:

```
CORES                                d---
  2019-09-02 01:01
DEMOS                                d---
  2019-09-02 01:01
LICENSE.MD                           ----
  2019-09-02 00:07   5243
README.MD                            ----
  2019-09-02 00:07   1427
TBBLUE.FW                            ----
  2019-09-02 00:07   172032
TBBLUE.TBU                           ----
  2019-09-02 00:07   475648
```

You can immediately notice two things: First the addition of a column made from four characters at the rightmost side of the screen and secondly that every entry now occupies two lines with the second containing a date, a time and a number (not in all cases). Let's start from the second line. Two types of information is available there; *when* the file or folder was created and *what's its size* (in bytes). The first line is the file itself (or the folder) while the rightmost column describes the file's *attributes*. The d you can see in some entries is the *directory attribute* which designates a folder. Folders as far as the filesystem is concerned are special files without size. In the shorter form of CAT we saw previously, this is displayed as <DIR>. There are many more attributes to examine which we will look at later.

You may have noticed that the display gets very cluttered when using the EXP modifier especially if there are a lot of files with long names as the screen normally fits only 32 columns. If you follow the note in the beginning of this chapter and use 64 or 85 column modes. you'll see the situation improves. Switch to 64 column or 85 column mode, rerun CAT EXP and you will get something similar to this:

```
DEMOS                          d---  2019-10-22 20:28
DOCS                           d---  2019-10-22 20:28
DOT                            d---  2019-10-22 20:28
GAMES                          d---  2019-10-22 20:28
MACHINES                       d---  2019-10-22 20:28
NEXTZXOS                       d---  2019-10-22 20:28
RPI                            d---  2019-10-22 20:28
SRC                            d---  2019-10-22 20:28
SYS                            d---  2019-10-22 20:28
TMP                            d---  2019-10-22 20:28
TOOLS                          d---  2019-10-22 20:28
CHANGELOG                      ----  2019-10-22 10:10  2940
CONTRIBUTING.md                ----  2019-10-22 10:10  9662
LICENSE.MD                     ----  2019-10-22 10:10  5186
README.MD                      ----  2019-10-22 10:10  1401
TBBLUE.FW                      ----  2019-10-22 10:10  172032
TBBLUE.TBU                     ----  2019-10-22 10:10  475648
CORES                          d---  2019-10-22 20:28
test.bas                       -a--  1980-00-00 00:00  212
test.l2                        -a--  1980-00-00 00:00  49280
test.sl2                       -a--  1980-00-00 00:00  49280
test.l2.bak                    -a--  1980-00-00 00:00  49280
BUBBBOBB.TAP                   ----  2005-04-05 13:07  53224
Bubble Bobble (1987)(Firebird)(48K-128K).tap
```

*Fig. 19 – CAT EXP output in 85 columns*

Similarly, the output will be even more pleasant at 64 columns:

```
DEMOS                          d---  2019-10-22 20:28
DOCS                           d---  2019-10-22 20:28
DOT                            d---  2019-10-22 20:28
GAMES                          d---  2019-10-22 20:28
MACHINES                       d---  2019-10-22 20:28
NEXTZXOS                       d---  2019-10-22 20:28
RPI                            d---  2019-10-22 20:28
SRC                            d---  2019-10-22 20:28
SYS                            d---  2019-10-22 20:28
TMP                            d---  2019-10-22 20:28
TOOLS                          d---  2019-10-22 20:28
CHANGELOG                      ----  2019-10-22 10:10  2940
CONTRIBUTING.md                ----  2019-10-22 10:10  9662
LICENSE.MD                     ----  2019-10-22 10:10  5186
README.MD                      ----  2019-10-22 10:10  1401
TBBLUE.FW                      ----  2019-10-22 10:10  172032
TBBLUE.TBU                     ----  2019-10-22 10:10  475648
CORES                          d---  2019-10-22 20:28
test.bas                       -a--  1980-00-00 00:00  212
test.l2                        -a--  1980-00-00 00:00  49280
test.sl2                       -a--  1980-00-00 00:00  49280
test.l2.bak                    -a--  1980-00-00 00:00  49280
BUBBBOBB.TAP                   ----  2005-04-05 13:07  53224
Bubble Bobble (1987)(Firebird)(48K-128K).tap
```

*Fig. 20 – CAT EXP output in 64 columns*

It's evident that the columns are really 4 and they only get broken down in two lines in order to fit. Let's now examine the use of the – switch. If you type:

```
CAT -
```

Your display now will look similar to this:

```
CORES    .                    <DIR>
DEMOS    .                    <DIR>
DOCS     .                    <DIR>
DOT      .                    <DIR>
GAMES    .                    <DIR>
MACHINES.                     <DIR>
NEXTZXOS.                     <DIR>
RPI      .                    <DIR>
SRC      .                    <DIR>
SYS      .                    <DIR>
TMP      .                    <DIR>
TOOLS    .                    <DIR>
LICENSE  .MD                      6K
README   .MD                      2K
TBBLUE   .FW                    168K
TBBLUE   .TBU                   465K

 1887M free

0 OK, 0:1
```

As you can see, filenames are now clearly separated at the 9th character by a dot followed by a 3 letter *type*. In order to demonstrate what happens with a larger filename we could write a simple program and save it as follows:

```
   10 PRINT "Hello World"

SAVE "This Is A Hello World Program.bas"
```

Then try both CAT and CAT - as follows:

```
CAT - "th*.bas": CAT "th*.bas"
```

(Here we're also demonstrating the use of *wildcards* for the first time). Your display will then be:

```
THISIS~1.BAS                          1K

 1887M free
This Is A Hello World Program.ba
s                                     1K

 1887M free

0 OK, 0:1
```

you'll notice that the *long filename* This Is A Hello World Program.bas got truncated to its first 6 characters after trimming all space characters followed by a tilde ~ character and the number 1. This is to help differentiate from other files with *long filenames* that look alike in the first 8 characters of their *filename* (omitting spaces). To demonstrate this, type:

```
SAVE "This Is A Hello United Kingdom
Program.bas"
```

and

```
SAVE "This Is A.bas"
```

followed by

```
CAT - "th*.bas"
```

The resulting display will now be:

```
THISIS~1.BAS                          1K
THISIS~2.BAS                          1K
THISISA .BAS                          1K

 1887M free

0 OK, 0:1
```

As you can see a ~2 was added to the This Is A Hello United Kingdom Program.bas *filename* when it was shortened otherwise you couldn't differentiate it from the This Is A Hello World Program.bas as they both share the same starting characters. As a matter of fact *NextZXOS* when faced with a lot of similar *filenames* will keep adding consecutive numbers truncating the original *filename* further until all the files are displayed in short format. If you now use CAT with EXP you'll get to see a number of things. First, if you don't have a *Real Time Clock module* installed, you will see that all the files you just saved have the same date and time on them and secondly that in the second column, the second character from the left has turned into a from a single dash (-). This signifies that the *archive attribute* has been set. CAT becomes more powerful with the use of *wildcards*, allowing us to get a list of only the files we're interested in, omitting all others that may clutter our display. For example:

```
CAT "*.tap"
```

will show us all the .tap *format tape image files,* we have stored in the current *drive* and *folder*.

Thus far we have only displayed the ability to list files contained within the *current drive* and folder, however CAT can display *files* in different *drives*, *folders*, *user areas* or a combination of the above (when the combination is supported by the *filesystem* of the *drive*). We can instruct CAT to produce listings of *files* and *folders* inside *drives* other than our *current drive* or *folder* or even *user area* without having to change our *default filespec* to that specific area. We'll cover the subject of changing the *default filespec* shortly so for now here are some examples:

| | |
|---|---|
| CAT "m:" | Displays a list of all files in drive m: |
| CAT "2m:" | Displays a list of all files in user area 2 of drive m: |

| | |
|---|---|
| CAT "2m:*.bas" | Displays a list of all files ending in .bas in user area 2 of drive m: |
| CAT "c:/nextzxos/" | Displays the contents of folder nextzxos found on drive c: |
| CAT "c:/nextzxos/e*.*" | Displays all files whose filename starts with the letter e in the folder nextzxos on drive c: |

CAT has two *aliases* in *NextBASIC*: DIR and LS. Both follow the exact same syntax so all the above applies to them. There are also two *dot commands* .ls and .lstap which are available on both *NextZXOS* proper and the *48K Basic mode* available from the *Startup menu*. They replicate CAT and the combination of .tapein[15] and CAT "t:" respectively. .ls has a lot more options available than CAT which can be seen once you type:

```
.ls --help
```

which will give you about 3 screens full of available options! For most purposes however it is used in the same manner as CAT filespec-wise. .ls does not require the *filespec* to be enclosed in double quotes if there is no *drive* specified (*drives* contain colon characters and both Sinclair as well as *NextBASIC* consider this as a statement separator and will complain). One major difference in the way .ls displays the files versus how CAT displays the files is that it uses the short format; ie. it's closer to giving CAT - than just plain CAT. Similarly, .lstap provides extra information than CAT "t:" provides as you will see by giving:

```
.lstap --help
```

.lstap is particularly useful in 48K Basic mode as there is no CAT "t:" equivalent in that version.

## Drive, Folder and User Area navigation and management

One of the major features of any operating system such as *NextZXOS* is the organisation and management of files within the capabilities of its supported filesystems. In earlier times, such as when the predecessor models of the ZX Spectrum Next were first available, file storage needs were not as pressing as they are today.

Storage media couldn't really hold a lot of information and even program sizes were tailored to the memory available to the computers of the era. Operating systems in other words, weren't really needed unless one had very important business files to manage. As time went on and computer capabilities grew, the few files that could fit on a tape or a microdrive cartridge became the tens that could fit on a floppy disk while today with the capacities of storage media skyrocketing we have to manage tens or even hundreds of thousands of files. Compare a microdrive cartridge that held 90 KBytes of data which was a massive capacity for the times, to your System/Next™ distribution that can hold 176 million times as much.

Early on, once the first disk based systems became available, the need to organise files in a more logical way was recognised and the first type of grouping of files was realised in the form of 16 user areas (numbered from 0 to 15). User areas served other needs as well but for a machine like the ZX Spectrum +3 that introduced it to the ZX Spectrum line, it was a means to gather together files. User areas are more than adequate for limited capacity storage media but wholly inadequate for larger media like the multi-megabyte hard drives that followed.

To that effect the concept of a folder (also known as a directory) was introduced which in itself can hold other folders in a nested organisational chain. This structure is called a directory tree (it's really an inverted tree with the root of it sitting at the top).

The FAT filesystem used on your System/Next™ distribution is a prime example of that organisation. It's obvious that with folders being nested, constantly writing commands like SAVE or LOAD that includes the length of any number of folders in addition to the file's name itself can be very copious. To that effect apart from the commands that deal with the

15 *.tapein is a dot command utility that lets NextZXOS assign a virtual tape image to the t: drive instead of the real tape*

creation and deletion of folders, *NextZXOS* provides us with commands to navigate the filesystem's directory tree. The filesystem navigation and management commands are:

### MKDIR

**MKDIR** (for MaKe DIRectory) creates a folder on a drive that supports it. It's syntax is as follows:

**MKDIR** *filespec*

where *filespec* follows the syntax already discussed in the *Filenames section* of this chapter using the first two parts that make up a filename: *Drive* and *Folder*. In the absence of a *drive* and an initial *folder separator* character, the folder you're creating will be created under the current folder and drive you've set. You can mix the *folder separators* \ and / without a problem when structuring the *filespec*. An attempt to create a folder with **MKDIR** in a filesystem that doesn't support it will report a **Non Implemented, 0:1** error.

If you are using **MKDIR** with a depth of folders greater than one, the folder name you're using must already exist otherwise you will receive an **Invalid path, 0:1** error. Here are some examples to illustrate:

| | |
|---|---|
| **MKDIR "/codes"** | Creates a folder named **codes** under the current drive's root folder. |
| **MKDIR "/codes/codes"** | Creates a subfolder named **codes** under the current drive's root folder inside the **codes** folder. If there is no folder named **codes** under the root folder, the command will fail. |
| **MKDIR "d:/test"** | Creates a folder named test under the d: drive's root folder |
| **MKDIR "d:test"** | Creates a subfolder under the **d:** drive's last changed-to folder. |

The last example is very interesting as it introduces the concept of *current folder per drive*. Indeed, *NextZXOS* maintains a list of which *folder* was *last changed to* on each *drive* and will switch you to that if you don't explicitly define a full pathname and only a *drive*. This will become very useful when copying as we will see later on.

There is a dot command equivalent of **MKDIR**, which shares its name apart from the dot prefix: **.mkdir**. It accepts two more, mutually exclusive options over **MKDIR**: --verbose and **--help** otherwise it's syntactically the same. As with most dot commands if there's no *drive* inside the *filespec* the double quotes enclosing it are optional.

### RMDIR

**RMDIR** (for ReMove DIRectory) removes an *empty folder* from a *drive* that supports *folders*. Its syntax is as follows:

**RMDIR** *filespec*

where *filespec* is as discussed in **MKDIR** above. **RMDIR** protects you from accidental deletion of files that can be contained within the folder by returning a **Dir full, 0:1** error if even one file or another folder is contained within. You will need to first remove all the files and subfolders located inside the folder before **RMDIR** allows you to remove the folder. Wildcards do not work with **RMDIR**; you cannot use **RMDIR "*"** and expect to remove all folders under the location you are in. Any attempt to do so, will return a **Bad filename, 0:1** error.

Finally, if you attempt to use **RMDIR** with a *folder* that doesn't exist, you will receive a an **Invalid path, 0:1** error.

**.rmdir** is **RMDIR**'s dot command equivalent. It is a bit more destructive than **RMDIR** as it allows the deletion of parent folders with the addition of optional switch **--parents**, however, it too, checks for data inside the folders slated for deletion and will return an error if data exists. With the exception of the optional switches **--parents** and **--help**, syntax for both **RMDIR** and **.rmdir** is the same.

## CD

**CD** (for Change Directory) changes the current *drive* and/or *folder* (for *drives* that support *folders)* or current *drive* (for *drives* that do not). **CD**'s syntax is as follows:

**CD** *filespec*

where *filespec* consists of either one or two of the first two parts of a filename (*Drive* and *Folder*) for *filesystems* that support *folders* (*FAT16*, *FAT32*) or of just the *Drive* for *filesystems* that do not (+*3DOS*, *IDEDOS*). Setting just the *current drive* with **CD** is functionally equivalent to using **SAVE**, **LOAD** etc with just the *drive* as the *filespec*. Unlike *folders*, there is no way of setting a *user area* as the default one so if you need to address it you must do so explicitly through the *filespec;* for example add a **3m:** prefix to *filenames* for files in the *user area* **3** of drive **m:**. **CD** works with *wildcards* by matching to the first folder in order it finds them and change to that.

**CD** also accepts three *filespec shortcuts*: **.** (single dot), **..** (double dot) and one of the following / or \ (forward or backward slash). As we mentioned earlier in the chapter, single dot means: *This* folder, double dot means: The folder *one level up* and either slash on their own means: The *root folder* of the *current drive*. Single and double dot entries *do not exist* on the *root folder* and therefore you cannot use the *shortcuts* there.

Using a combination of the double dot and slash *shortcuts*, **CD** can also easily traverse the folder tree horizontally at the same level without having to write the entire path that precedes the level you're currently in. Obviously that doesn't make sense at the first level under the root as it would involve much more typing than the slash character alone but it works nonetheless!

Assuming a structure like the one in your **System/Next**™ distribution as partly displayed in the figure below, lets provide some examples of horizontal and vertical navigation.

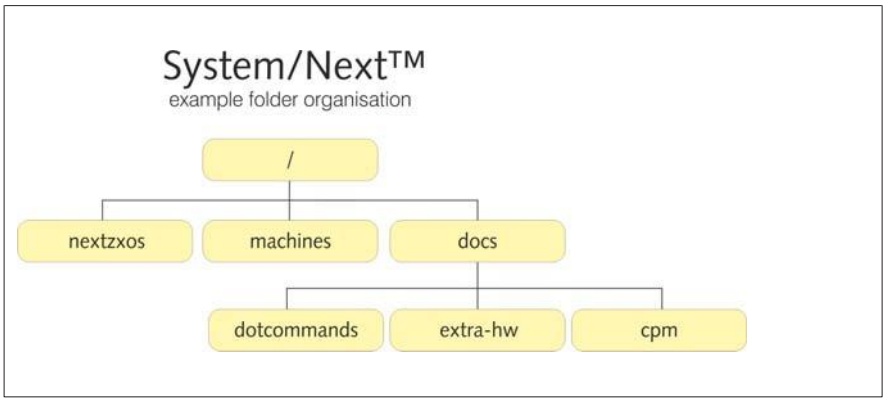



*Fig. 21 – Folder tree navigation*

Let's agree that we're located in the / of drive **c:** and we want to first go to **c:/docs/cpm** and then go to **c:/docs/extra-hw** before returning to / again.

We could use one of the following sequences:

```
CD "docs"
CD "cpm"
```

and then

```
CD ".."
CD "extra-hw"
```

and finally

```
CD "../.."
CD "../.."
```

or alternatively:

```
CD "c:/docs/cpm"
CD "c:/docs/extra-hw"
CD "../.."
CD "../.."
```

However it's much less typing to just do:

```
CD "/docs/cpm"
CD "../extra-hw"
CD "/"
```

It's easy to see that the navigational shortcuts are quicker. The *dot command* equivalent of CD is .cd with the optional switch --verbose which performs the functions of both CD and PWD (see below) in order. A small deviation from the syntax of CD is that it allows specific shortcuts to navigate quickly to the top folder of a deeply nested hierarchy.

These are:

| | |
|---|---|
| .cd ... | Functionally equivalent to two successive CD ".." commands |
| .cd .... | Functionally equivalent to three successive CD ".." commands |
| .cd ..... | Functionally equivalent to four successive CD ".." commands |

### PWD

PWD (for Print Working Directory) prints the current *drive* and *folder* to the screen or an optional stream number. PWD's syntax is as follows:

PWD *[#n]*

In a *NextZXOS* context PWD is very useful, however you cannot assign its output to a *NextBASIC* variable that easily for use inside our programs. In order to do that, one should be a little creative (skipping ahead to the next chapter) and use the optional *stream* parameter in a manner identical to the trick we used to get time from our RTC back in *Chapter 17*. Type:

```
DIM d$(255):OPEN #2,"v>d$":PWD #2:CLOSE
#2: PRINT d$
```

with which we define a fixed size string variable d$, then open stream 2 and assign it to channel V which redirects its output to d$. We then invoke PWD with output redirection to stream 2 which in essence takes its normal screen output and via *channel* v sends it to d$, before closing the stream and printing d$. We did exactly what PWD would do normally (that is print the working directory on the screen) but also managed to store it in a variable for use later.

PWD doesn't have a dot command equivalent with the same name. Instead you only need to use .cd --verbose without a *filespec*. The example above therefore becomes:

```
DIM d$(255):OPEN #2,"v>d$":.cd --verbose:
CLOSE #2: PRINT d$
```

You may notice that there's no *stream* defined after .cd --verbose and that's because you don't need it as *stream* #2 is the screen anyway! It's obvious that the same applies to PWD above but PWD does offer the ability to redirect to a *stream* and that illustrated that fact quite nicely. As a matter of fact, you can completely omit the *stream* from the PWD statement in the previous example and it will function in the same manner; you will see why in the next chapter.

## Managing files and their attributes

In our examples in this chapter we have managed to clutter our drives with lots of copies of the same programs. This may be desirable at times but sometimes we may want to keep slightly altered versions of the same program in different places (for example to keep a type of version history) but we may not have the organisation of the folders we'll store the files in when we start working.

Other times we may want to get rid of some files we've created for any number of reasons, or rename a file from a throwaway name like for example test.bas to something more meaningful and finally we may want to move some files from one place to another when done with them. *NextZXOS* provides us with all these facilities in the form of the COPY, ERASE and MOVE commands and their dot command equivalents .cp, .rm and .mv.

We'll examine these below and additionally find how to modify file *attributes* (what is displayed as the second column in the CAT EXP command's output) again via a special version of MOVE and its dot command alternative .chmod. There is one more function provided by *NextZXOS* in regards to files and that's directly accessing its contents. This however requires the use of *Channels* and *Streams* and is therefore covered in the next chapter.

### COPY

COPY does as its name implies; Copies a file from a location to another location. Its syntax is quite simple:

COPY *source* TO *destination*

A few notes, regarding the differences between *source* and *destination* parameters are:

First and most importantly, *source* can use *wildcards* while *destination* cannot. In other words you can write:

```
COPY "c:\*.bas" TO "m:"
```

but you cannot write:

```
COPY "c:\*.bas" TO "m:\*.bas"
```

or

```
COPY "c:\*.bas" TO "m:\a*.bas"
```

as any attempt to do so will generate a Destination cannot be wild, 0:1 error.

Secondly, copying files between *filesystems* with different capabilities will perform some form of translation to the filenames. To give an example with two files named raycaster.bas (longer than 11 characters) and ..later.bas (starting with two dots) on drive c: doing:

```
COPY "c:\*.bas" TO "m:"
```

will change the filenames to raycas~1.bas and later.bas as the *RAMdisk* is a *+3DOS drive* and as such accepts only 8+3 filenames.

Thirdly, the *destination* is not checked for if the files being copied already exist. So if you perform the above operation twice, each time COPY will replace the files on the *destination* without creating backup files except if the file named the same in the destination has the *protected attribute* set. To demonstrate let's skip a bit ahead and introduce you to an attribute setting command. Type the following:

```
COPY "c:/nextzxos/pisid.*" TO "m:"
MOVE "m:PISID.BAS" TO "+p"
COPY "c:/nextzxos/pisid.*" TO "m:"
```

The first COPY operation will succeed while the second COPY operation will fail. In the case of a mass COPY if the operation fails for any file, it will fail for all remaining files, so keep that in mind.

COPY does not work between a disk and a tape; doing for example:

```
.tapeout "test.tap"
COPY "m:*.bas" TO "t:"
```

will fail with a Destination must be path, 0:1 error. Note above the use of the .tapeout dot command which we will cover later on; it just allows us to substitute a *tape image* file for an actual tape. To perform the above function we will need to do the following:

```
.tapeout "test.tap"
LOAD "m:hello.bas"
SAVE "t:hello.bas"
```

and verify the output with .lstap we covered earlier:

```
.lstap "hello.tap"
```

(or alternatively not use .tapeout and .lstap at all and save onto an actual tape, in which case we'd use VERIFY to check if the file was actually written)

There is a special version of COPY where the *source* file is stripped of all *control codes*, just maintaining *End-Of-Line* characters (CR, LF or the combination of both – See *Appendix A* for all Control Codes). It exists as either shortcuts SCREEN$ and LPRINT in lieu of destination -or- as any *stream* that can be attached to a *channel*. The SCREEN$ shortcut gets any file and prints it on screen while the LPRINT shortcut gets any file and sends it to a ZX Printer or compatible. A good way to test the functionality is to check some of the documents in c:/docs. For example to see the pinouts of the Next board you can type:

```
COPY "c:/docs/extra-hw/pinouts/pin*.txt"
TO SCREEN$
```

while if you do:

```
COPY "c:/docs/extra-hw/pinouts/pin*.txt"
TO LPRINT
```

the file will be sent straight to the printer! SCREEN$ and LPRINT are shortcuts for their respective *streams* (as you will see in the next chapter). Although there are no shortcut keywords for other *streams*, if the *destination* is set to any *stream*, COPY's behaviour will be identical to what we just saw.

The *dot command* equivalent for COPY is .cp and its syntax is similar with the exception of the --force switch which allows overwriting of files without prompt. .cp CANNOT currently address *+3DOS/IDEDOS drives* so it should be only used on *FAT partitions* on the SD Card.

## ERASE

Files can be deleted from a drive using the ERASE command. Its syntax is as simple as one would imagine:

ERASE *filespec*

where *filespec* follows the same conventions as CAT meaning that just like CAT, you can use the *wildcards* * and ? to identify a group of files, or you can specify the filename in full (including optional Drive and/or User Area and Path) if you only want to get rid of one par-

ticular file. ERASE offers you some form of protection if your *filespec* contains *wildcards* in the form of a question in which you will have to answer with a Y on the keyboard to continue or with N to stop, but offers no protection if you specify a single filename, which will immediately be erased from the drive – so exercise caution! If, for example, you wanted to delete a file from drive m: called FRED.BAS, you would use:

```
ERASE "m:fred.bas"
```

If drive m: has already been set as the *default drive* (by either using SAVE, LOAD... or even CD), then you don't need to include the m: at the start of the filename. It doesn't hurt to include the drive anyway, and with as powerful a command as ERASE is, you might feel safer if you do. To erase all the files on drive d: you would use:

```
ERASE "d:*.*"
```

Before doing this, *NextZXOS* will ask for confirmation by printing

```
Erase d:*.* ? (Y/N)
```

on the bottom of the screen and assuming that you really mean to wipe all the files from the disk in drive d:, you would then type Y.

If you attempt to delete a single file (or a group of files using *wildcards*) while there are no files on the drive that match the *filespec* a File not found error will be displayed.

The dot command equivalent to ERASE is called .rm (from remove) and its syntax follows that of ERASE with the exception of two switches namely --verbose and --help.

MOVE

MOVE is a very powerful command. It performs a total of five functions: *moving* and *renaming* files, *changing* file *attributes* and manually *mounting* and *dismounting drives*. Since there are separate sections for the last three functions; we'll cover only the first two here. For *moving* and *renaming*, MOVE's syntax is:

MOVE *source_filespec* TO *destination_filespec*

where *source_filespec* and *destination_filespec* follow everything discussed in the *Filenames section* earlier with the following considerations:

- You cannot use *wildcards* in either the *source* or the *destination*. This means that both *source* and *destination* have to be *complete filenames*.
- You cannot perform a MOVE operation between drives

Let's examine what will happen in the first case. Assuming you have 3 NextBASIC files, named HELLO1.BAS, HELLO2.BAS and HELLO3.BAS in drive m: (in the default *User Area* 0) and you want to move them to *User Area* 1, typing as you would probably expect:

```
MOVE "*.bas" TO "1:"
```

will fail with Bad Filename, 0:1. To perform this you should actually do:

```
COPY "*.bas" TO "1:"
```

followed by

```
ERASE "*.bas"
```

In the second case (and since we now learned our lesson we won't be using wildcards) attempting to MOVE one file between drives like so:

```
MOVE "c:/test.bas" TO "d:/test.bas"
```

will fail with No rename between drives, 0:1. To perform this you should actually do like above:

```
COPY "c:/test.bas" TO "d:/"
ERASE "c:/test.bas"
```

As you probably have already figured out, moving and renaming files is basically the same procedure and since we have to write an entire filename in both source and destination we can change it at the same time!

```
MOVE "hello1.bas" TO "c:/bak/hello.bak"
```

both *moves* locations and *renames* hello1.bas.

Imagine we have saved a file called FRED, and then after working on it and saving a new version with the same name, realised that we had made a terrible mistake and would like to recover the last version. This would be possible using the commands:

```
ERASE "fred"
MOVE "fred.bak" TO "fred"
```

If a file you're moving or renaming already exists (or rather another file with the same name) at the intended destination, MOVE will fail with an Already exists, 0:1 error.

MOVE's dot command alternative is .mv and unlike other dot command alternatives we've examined so far, its renaming and moving capabilities far exceed those of MOVE's. It allows *operations* across different drives, *interactive* or *automatic overwriting* of already existing files as well as the full use of *wildcards*. It's syntax is:

.mv [OPTION] [-T] *source destination* –or–
.mv [OPTION] *source* DIR –or–
.mv [OPTION] -t *DIR source*

Where *source* and *destination* can be any valid *NextZXOS filespec* (including *wildcards*) and *DIR* is any valid *folder* . *Source* or *Destination filespecs* with trailing slash characters (/ or \) are considered to be folders. As .mv has numerous options, they are listed in the table below to help you better understand what it can do. In general when you have a large quantity of files to be moved or renamed it's better to use .mv over MOVE.

| Option | Alt Option Syntax | Description | Notes |
|---|---|---|---|
| -b | | Makes backup of existing destination | |
| -f | --force | Do not prompt for overwrite | Of these three options, the last in order is the one that takes effect |
| -i | --interactive | Prompt for overwrite | |
| -n | --no-clobber | Do not overwrite | |
| | --strip-trailing-slashes | Remove slashes from names | |
| -S | --suffix=SUFFIX | Override default backup suffix with SUFFIX | |
| | --system | Match system files to source | |
| -t DIR | --target-directory=DIR | Move everything in source to folder DIR | |
| -T | --no-target-directory | Treat destination as a normal file | |
| -u | --update | Move only if source is newer than destination or destination doesn't exist | |
| -v | --verbose | Explain what is being done | |
| -h | --help | Prints this list of options | |
| -v | --version | Prints the version of .rm and exits | |

*Table 16 – .rm options*

## File attributes

As mentioned in the previous section, MOVE has another use besides renaming and moving files and that is to change a file's attributes. Attributes are bits of information associated with a file that tell you (and the computer) a little more about it. You already saw in the CAT EXP and ERASE examples how attributes appear to you and how they can affect your files. There are three attributes that can be changed plus one more that is automatically managed: *write protection*, *system status* and *archive*. The most useful attribute is, as

we've seen already, *write protection*. Once a file's write protection attribute has been set, it will not be possible to erase it (or save a file with the same name) until you remove it.

MOVE's syntax for *attribute* changing is a bit different from the one used for renaming/moving:

MOVE *filespec* TO +/*-attribute*

Where *filespec* CAN include *wildcards* unlike the previous case, and *attribute* is one of the following letters: **p**, **a** and **s** used with either a + or **-** prefix. The prefix serves as a set (for +) and unset/clear (for -). **p** is short for **protection**, **a** is short for **archive** and **s** is short for **system.**

Write protection is the most useful attribute for *NextZXOS*. Try:

```
MOVE "hello.bas" to "+p"
```

If you now try:

```
ERASE "hello.bas"
```

ERASE will fail with a **File is read only** error.

To switch write protection off type:

```
MOVE "hello.bas" TO "-p"
```

and you'll be able to erase the file as before.

As mentioned, we can use wildcards when changing attributes. As an example, to make all the files on drive **m:** write protected, you would type:

```
MOVE "m:*.*" to "+p"
```

As always, the drive letter can be omitted if it is the current default drive.

You can repeatedly switch attributes on or off without causing an error, so if you set write protect on a file that has already got write protection, it will just stay protected.

The second attribute we mentioned is the *system status attribute*. This is really provided just to be compatible with other *CP/M* based computers, however, if you do set a file's *system attribute* to *on*, you will see that the file no longer appears in the list when doing a normal CAT. It will appear however when using **CAT EXP** with an **s** marked in the second column and when using **.ls**. Try the following:

```
MOVE "hello.bas" TO "+s"
CAT
CAT EXP
LOAD "hello.bas": RUN
```

As you can see **hello.bas** became invisible to **CAT** but you can still **LOAD** it properly if you know its name. Bear in mind that you cannot have two files on the same disk with the same filename and different system status attributes; so if you try to create or copy a file onto a disk where a file of that already exists (but is hidden from **CAT**), then the previous file will be deleted, unless of course its *write protect attribute* is set.

The final attribute you can change is known as the *archive attribute*. In an expanded catalogue, it shows up as **a**. On other systems the *archive bit* is cleared when a copy operation has been performed, but that doesn't happen on NextZXOS. NextZXOS automatically sets the archive bit when saving on a FAT driver but doesn't do so on IDEDOS/+3DOS drives. It is therefore of no practical use and is only provided for file compatibility with CP/M.

If you try to use any letter other than **a**, **s** or **p** in setting or resetting attributes, or if the attribute option string is not two characters long, then you will receive an **Invalid attribute** error.

The *dot command* that handles attributes is **.chmod** and has a bit of a different syntax than **MOVE** as it accepts four attributes **r**, **h**, **s** and **a**, for read-only, hidden, system and archive. The first is in essence the same as **p** for **MOVE** while **h** doesn't exist on *NextZXOS* (setting the system attribute makes it also hidden by default) but it does exist as an attribute on *FAT drives*. Trying:

```
.chmod TBBLUE.FW -h
```

you will see that nothing has changed when doing **CAT EXP**. If you however take your SD Card to a PC, you will be able to see the file again there.

## The RAMdisk

You may have been wondering what point there is in storing information in the RAMdisk (**m:**) as it will be lost once the ZX Spectrum Next is switched off. Well, perhaps its most obvious use is to store chunks of *NextBASIC* program (or routines) which can be merged (using **MERGE "m:filename"**) into a smaller program, in sequence. This makes it possible to write about 90K of *NextBASIC* code, and hold it in the machine, without going into the more complicated **BANK** commands. Another little less obvious use is to store temporary files there that won't be needed when your program finishes. Memory is the fastest medium on your ZX Spectrum Next and quick access to files may be beneficial.

As we saw in *Chapter 17*, one of the more interesting uses of the *RAMdisk* is in animation, where a series of pictures can be defined by a slow *NextBASIC* program, stored in drive **m:**, then called back to the screen at high speed. Obviously **BANK** is still the preferred way to do it, but for quick jobs that use *Layer 0* it's a quick and easy method!

## Drive and Partition Management

We've talked about physical devices and virtual devices; we've also talked about the automounting features of *NextZXOS* but we haven't truly explored how the system manages *storage devices* and assigns them to *drives*. *NextZXOS* provides us with four commands to help us list and manage disks and drives. The drive and partition management commands are: **CAT TAB** that lists the physical storage devices attached to the system and what partitions they contain; **CAT ASN** that lists all drive assignments to whichever partition or disk (basically listing what's mounted), **MOVE … IN** to assign any device/partition physical or virtual to a drive (mount) and **MOVE … OUT** to remove an assigned partition from a drive as well as **REMOUNT** that allows us to change system disks on the fly. *NextZXOS* also provides us with a way to create *virtual disks* of varying sizes in the form of two dot commands: **.mkdata** and **.mkswap**

## CAT TAB and CAT ASN

**CAT TAB** lists the storage devices currently connected to your ZX Spectrum Next and their partitions. It's syntax is:

CAT [*#n*] TAB

where *#n* is an optional *stream* to redirect the output to (e.g. to a file). On a standard ZX Spectrum Next with a single SD Card reader, giving

```
CAT TAB
```

will return:

```
MMC unit 0 (1024M)
MMC unit 5 (1024M)
5>1>NEXT                1024M FAT32
```

which illustrates also a point we made early in the chapter. Each SD Reader is assigned two device numbers (**0**,**5** and **1**,**6** for first and second SD Readers respectively) according to what partitions it holds. If for example we had eight partitions, seven IDEDOS and one FAT32 then our display would have been:

```
MMC unit 0 (1024M)
0>PLUSIDEDOS            64K sys
0>General             4096K data
0>CPM-A                320K data
0>CPM-B                512K data
0>CPMStuff             512K data
0>Dev                  256K data
0>Next                 320K data
0>"""""""""""""""""  10304K FREE
24 free partition entries
MMC unit 5 (1024M)
5>1>NEXT               1008M FAT32
```

CAT ASN on the other hand, displays which partition or disk is assigned to which drive. The syntax is similar to CAT TAB:

CAT [*#n*] ASN

where, again, *#n* is an optional *stream* for the output to be redirected to. On a standard ZX Spectrum Next with a single SD reader and prepared *CP/M* (whose virtual drive a: as we have discussed would be already automounted), giving:

```
CAT ASN
```

would produce the following output:

```
A: ---Mounted FS---
C: 5>1>NEXT
M: 4>RAMdisk
```

If you are asking what happened to the *IDEDOS partition* we displayed earlier, it's not mounted because *IDEDOS partitions* do not auto-mount. To mount them (or any other partition or virtual/physical disk) you will need to employ the following commands:

## MOVE ... IN, MOVE ... OUT and REMOUNT

In order to assign (mount) a disk/partition or virtual/physical disk to a drive you need MOVE ... IN. Its syntax is as follows:

MOVE *drive* IN *mount_point*

where *drive* is any valid *NextZXOS* drive (a: to p:) and *mount_point* is either a *device>[partition][>][partition_name]* or a *filespec* of a *virtual disk*. Devices that don't have partitions are written as *X>* where *X* is the device number, while devices that have partitions are written as *X>Y>[partition_name]* where *Y* is the *partition number* for *FAT partitions* and *X>partition name* for *IDEDOS partitions*. In the case of *IDEDOS* partitions the number can be totally omitted as well if on device 0. Assuming that we had unmounted the *RAMdisk*, in order to mount it again in some other drive, we'd need to do:

```
MOVE "o:" IN "4>"
```

Notice that there's no partition number following the 4> as the *RAMdisk* has no partitions. To mount a +3 disk image named mike.dsk located in c:/images/ into drive b: we would need to:

```
MOVE "b:" IN "c:/images/mike.dsk"
```

Whereas to mount an *IDEDOS* partition (for example one of the ones we examined earlier) you would have to:

```
MOVE "e:" IN "0>CPMSTUFF"
```

or

```
MOVE "e:" IN "CPMstuff"
```

Attempting to mount a drive that's already assigned will produce the error **Already exists, 0:1**. In order to do that, you'll first need to unmount the drive with **MOVE ... OUT**. The syntax is even simpler:

**MOVE** *drive* **OUT**

So to unmount the disk image from **b:** we just need to give:

```
MOVE "b:" OUT
```

You cannot unmount the **c:** drive and attempting to do so will report an **In use, 0:1** error. You can however temporarily eject it (for example to write to it or just change it to a different version of *NextZXOS*, or even a game). Doing that without powering down or just arbitrarily, can damage your card beyond repair so you must be VERY careful. Since the potential for damage is great, *NextZXOS* has a special command to address that specific need called **REMOUNT**. Remount is given without any parameters and upon invocation it will prompt you to:

```
Remove/insert SD and press Y
```

Once you see the message you can eject your SD card, and when you reinsert it, press **Y**. *NextZXOS* will perform the same mounting procedure it performs on boot (for all drives) and your SD card contents will be safe!

### Virtual filesystem management – .mkdata and .mkswap

As we've already demonstrated, *NextZXOS* can read unprotected *+3DOS* and *IDEDOS virtual disks*, but how are these made? There are two ways to do it: We can either create them externally using special imaging software or right on *NextZXOS,* with the use of a specialised *dot command* called **.mkdata**. Its syntax is as follows:

**.mkdata** *filespec* [*size*]

where *filespec* must follow the requirements set forth in the *Filenames section* for legal filenames omitting the drive and *size* is an optional number from **1** to **16** (in Megabytes). Leaving *size* blank, will select the default size of 16 Megabytes. You can use ANY filename, however only filenames with a **.p3d** *type*, named as described in the *automounting section* earlier in this chapter and located inside **c:/nextzxos/** will be *automounted*. Here are some examples:

To make an 8 Megabyte *automountable* (as **a:**) *virtual disk*:

```
.mkdata /nextzxos/drv-a.p3d 8
```

To make a 16 Megabyte *virtual disk* that can be manually mounted in **c:/images/**:

```
.mkdata /images/disk.p3d
```

In order to make a *virtual disk* in a different *drive* you need to first change to it. For example:

```
CD "d:"
.mkdata /images/disk.p3d
```

will make a 16 Megabyte *virtual disk image file* named **disk.p3d** in **d:/images/**.

*NextZXOS* also supports *virtual memory* in the form of *virtual swap partitions*. These are similar to the *virtual disk images* with the difference that they cannot be mounted as drives. You can make *virtual swap partition images* with the **.mkswap** *dot command* which follows the same syntax as **.mkdata.**

**.mkswap** *filespec* [*size*]

To make an 8 Megabyte virtual swap partition image named swp-0.p3s you will need to give:

```
.mkswap /nextzxos/swp-0.p3s 8
```

Swap partitions named swp-0.p3s to swp-9.p3s which are present in the c:/nextzxos/ folder will be available for machine-code application programs to use (via the *IDEDOS API*).

## Printing

*NextZXOS* supports printing via ZX Printer, Timex Sinclair 2040 and compatibles like the Alphacom 32. It also supports printing via the *WiFi module* – if one is installed – and you have access to a Pipsta™ printer or a printer compatible with D. Rimron's *PrintShop* as found on: https://github.com/StalePixels/PrintShop.

To print a listing you only need the LLIST command while to print any string to the printer you need to use LPRINT. *Layer 0* and *Layer 1* screens can also be printed by using the COPY command given by itself with no options. In order to demonstrate this we will have to jump a bit ahead. Load one of the games from c:/games/Classic48/ (preferably one with a loading screen). Once you see the screen press the NMI button on the left side of your ZX Spectrum Next. A menu will appear. Using the cursor keys go to the *Screenshot* menu and press ENTER. Select *Standard* and Press ENTER. Press SPACE and type in a name (for example: test.scr) Press ENTER again and then press the reset button on the side of your computer or F4 on your keyboard. Re-enter *NextBASIC* and navigate to the location you were in. Then do the following:

```
LOAD "test.scr" SCREEN$:COPY
```

The screenshot will print on your printer!

Since you're undoubtedly observant you may have seen the *Print* item in the *Screenshot* submenu when you pressed the NMI button. That will do the exact same thing! But more on that in its own section below. There are also, other ways to print which we will examine in the next chapter.

## The SPECTRUM command

There is a command that's a bit of a jack of all trades; it can switch modes, load programs in various snapshot formats, change colour schemes, adjust the displayed columns for the editor and finally control and adjust the screensaver[16] function! Let's start with the simplest iteration of SPECTRUM which is the command without any options. This will take us into 48K mode preserving any *NextBASIC* program we have in memory but losing all Next mode features except for the dot commands which will be still available. If the program you have loaded in memory is using specialised *NextBASIC* features, LIST may produce gibberish (like graphics in the place of where commands would have been) and running it will probably produce a C Nonsense in BASIC error. Let's demonstrate. Type:

```
LOAD "c:/nextzxos/mounter.bas"
LIST
SPECTRUM
LIST
RUN
```

If you are in the standard ZX 48K mode, you will need to know the keywords, printed on your keyboard, but assuming you can find where CAT is (Press EXTEND then SYMBOL SHIFT and 9), type:

```
CAT
```

You will receive an O Invalid stream, 0:1 error. That's because 48K ZX Basic is unaware of any mass storage medium except for the ZX Microdrive and CAT is made to work with

16 *A screensaver is a protective function for your display. Some displays can damage themselves if they are displaying the same picture for a prolonged period of time. A screensaver program, produces movement on screen automatically after a period of inactivity to prevent that type of damage.*

that. In order to actually see what's on your drive, you will need the *dot command* equivalent of CAT, .ls. Indeed typing:

```
.ls
```

you will once again, see what's on your drive.

Once SPECTRUM is used to change to 48K Mode, you cannot return to the Next mode using a command (as SPECTRUM does not exist in 48K BASIC). Instead you will have to reset your machine, using either the Reset button on the side of the computer or by pressing NMI together with 1.

A more complex iteration of the command is the following:

SPECTRUM *filespec*

This command loads a snapshot file in the popular .z80, .sna, .snx[17], .p and .o formats and runs it. 48K, 128K as well as ZX80 and ZX81 snapshots are supported. Here are some examples:

To load the ZX81 classic 3D Monster Maze:

```
SPECTRUM "/games/zx81/3dmm/3dmonstermaze.p"
```

To load Pogie in Dreamworld Demo:

```
SPECTRUM "/games/next/pogie/pogie.snx"
```

To load Darkstar:

```
SPECTRUM "/games/classic128/
      darkstar.z80"
```

Two more specific variations of the basic SPECTRUM command are:

SPECTRUM LOAD

which switches to the ZX Spectrum 128K compatibility mode and invokes the Tape Loader and

SPECTRUM 48

which switches to the ZX Spectrum 48K compatibility mode, without any access to 128K hardware features. Both of those are mostly of use to the .TAP/.TZX/Tape Loaders included with *NextZXOS*.

To change colour schemes for the *NextBASIC Editor*, SPECTRUM can be used with one of the following modifiers: INK, PAPER, FLASH, BRIGHT and ATTR (which sets all the previous ones in one command). The syntax is as follows:

SPECTRUM *MODIFIER n*

where *MODIFIER* is one of INK, PAPER, FLASH, BRIGHT or ATTR and *n* is a standard colour from 0 to 7 when using the INK and PAPER modifiers, 0 to 1 for *disabled* or *enabled* when using the BRIGHT and FLASH modifiers, or calculated as: *(128*flash)+(64*bright)+(8*paper)+ink* for the ATTR modifier. Here are some examples:

```
SPECTRUM INK 4:SPECTRUM PAPER 0
```

or

```
SPECTRUM ATTR 4
```

17 *The .snx type is essentially the same as .sna but instructs SPECTRUM to load the snapshot using some Next mode settings (as for example ZXN DMA instead of Z80 DMA) as it prioritises features over compatibility.*

both set the *NextBASIC Editor* colours to green ink on black paper. You can see how the second one is derived by doing the following calculation: (128*0)+(64*0)+(8*0)+4

```
SPECTRUM PAPER 1:SPECTRUM INK 6
```

or

```
SPECTRUM ATTR 14
```

set the *NextBASIC Editor* colours to yellow ink on blue paper. Try to figure out how the second variation works!

The colour scheme applies to the standard 32-column editing mode as well as the hi-resolution 64/85 column modes. However, since *Layer 1,2* only allows 8 different colour schemes, the scheme used is the one with the same PAPER colour as standard mode.

SPECTRUM can also be used with the CHR$ modifier to set the number of columns in the NextBASIC editor. Its syntax is:

SPECTRUM CHR$ *n*

where *n* is one of 32, 64 or 85 for the available column modes. To switch for example to 64 column mode you should type:

```
SPECTRUM CHR$ 64
```

Attempting to enter a value other than 32, 64 or 85 as parameter will produce an Integer out of range, 0:1 error.

Finally, SPECTRUM used with the modifier SCREEN$ can control the *NextZXOS* screensaver behaviour. The syntax is as follows:

SPECTRUM SCREEN$ *n,t*

where *n* is the type of screensaver (0 = bouncing box, 1=blank screen) and *t* is the timeout in minutes from 0 to 127. If *t* is 0 then the screensaver is *disabled* until the next reset. The screensaver will activate (after the selected timeout) whenever the machine is waiting for a key to be pressed under the following circumstances:

- In menus, *Browser*, *Calculator*, *NextBASIC Editor* or while in the *Command Line*
- During INPUT statements
- During PAUSE 0 statements
- When NEXT #n,var is waiting for a keystroke from the K, S or W *channels*
- When executing machine-code software that uses the IDE_BROWSER call, or the IDE_STREAM_IN call (accessing K, S or W *channels*) or an IDE_BASIC call accessing the previously listed *NextBASIC* statements.

The screensaver will not activate when games are being run (unless they use the API calls listed above), or in 48 BASIC.

## Speed Control

The ZX Spectrum Next has a much faster *CPU* than its predecessors operating in one of the following speeds: 3.5MHz (same as the original ZX Spectrum), 7MHz, 14MHz and finally 28MHz. *NextBASIC* by default will set the CPU to execute at 3.5MHz, a setting which can be changed using either the left and right cursor keys while in any *NextZXOS menu* or directly from *NextBASIC* by using the RUN AT command. The syntax of the latter is as follows:

RUN AT *s*

where *s* is a number from 0 to 3 (0=3.5 MHz, 1=7 MHz, 2=14 MHz and 3=28 MHz). For example, to execute a program at 28 MHz begin the program with a:

```
1  RUN AT 3
```

## NextBASIC Editor and Program support commands

*NextZXOS* provides a few direct commands, that allow *NextBASIC* programmers to control both the appearance as well as the flow of their programs. These are:

ERASE [*first*, *last*]

erases all lines between *first* and *last* (inclusive) keeping any variable intact. ERASE on its own deletes the entire program (still keeping all variables intact) and unlike it's parameter version, can be included in a program (see the *autoexec.bas section* below for an example).

LINE *first, step*

renumbers the program starting at line *first* using a predefined *step*. Let's assume a small program:

```
10 FOR f=1 TO 10
20 PRINT f,
30 NEXT f
```

If we now give:

```
LINE 2,3
```

The program becomes:

```
 2 FOR f=1 TO 10
 5 PRINT f,
 8 NEXT f
```

It's obvious that we can pack as much "program" as we can in the amount of lines NextBASIC allows once our program is finalised. This should not be confused with the direct command

BANK *n* LINE *first, last*

which copies all lines in the main program between *first* and *last* to BANK number *n*. More on all bank-related commands can be found on *Chapter 23*.

LINE MERGE *first, last*

performs an even nicer optimisation to our typed programs, merging lines together to form a longer line, thus freeing lines for use. Assuming the program above, type:

```
LINE MERGE 2,8
```

the program then becomes:

```
 2 FOR f=1 TO 10: PRINT f,:
   NEXT f
```

Obviously LINE MERGE makes our programs less readable but let's us pack them even more allowing for even more line numbers to be freed.

BANK *n* MERGE

copies a banked program back into the main program (more details on *Chapter 23*) erasing everything that's already there with the same line numbers. For example, in the above LINE MERGE example, EDIT line 2 to be also line 4 by going over line number 2, deleting it and replacing it with a 4. Then do the following:

```
BANK NEW a
BANK a LINE 2,2
ERASE 2,2
LIST
```

and finally:

```
BANK a MERGE
```

You'll see that the line you erased with ERASE 2,2, is back into place

*NextZXOS* also provides one more command we've already seen but haven't sufficiently explained yet:

[BANK *n*] LIST [*#c*] [PROC *name*()]

which just lists the program (and optionally redirects its output to a stream) that's currently in memory. Optionally LIST can produce the list of the program that's currently in BANK *n*, or list a program whether banked or not starting with the procedure name()

%CODE

*NextBASIC* options are controlled by the special %CODE integer variable. This is reset to zero when a program is loaded/run.

Currently available options are:

| Bit | Use |
|---|---|
| 0 | if set, %RND n and RND(n) return values between 0..n, rather than 0..–1 |
| 1 | if set, the BREAK key is disabled |

## The Browser

In order to allow easier navigation of your files, *NextZXOS* comes with the *Browser*, a program that allows you to do so in a visual way. The *Browser* features the following:

- Easy navigation of drives and folders
- File management facilities: copying, erasing, renaming and moving of files
- Quick virtual disk mapping
- Automatic launching of known file types
- Extensible architecture for launching
- Cursor key or joystick navigation

The *Browser* is launched by using the EDIT key to bring up the *NextZXOS menus* or directly upon bootup by selecting the first entry in the *NextZXOS Startup menu*.

## The Browser Window

Once the menu is selected and ENTER is pressed, the screen changes to the *Browser* window containing a list of the files located in the *default drive and folder* (as set by the CD, LOAD, SAVE, MERGE or VERIFY commands in *NextBASIC*). Normally upon initial boot this will be c:/ but subsequent runs without a complete power down may show different locations reflecting the last drive and folder set as default. Note that you do not need to switch to *NextBASIC* to set a default drive and folder. Whatever you select with the *Browser* has the exact same effect for *NextBASIC,* as giving one of the aforementioned commands.

The *Browser* window consists of five separate areas, as seen in the figure below:

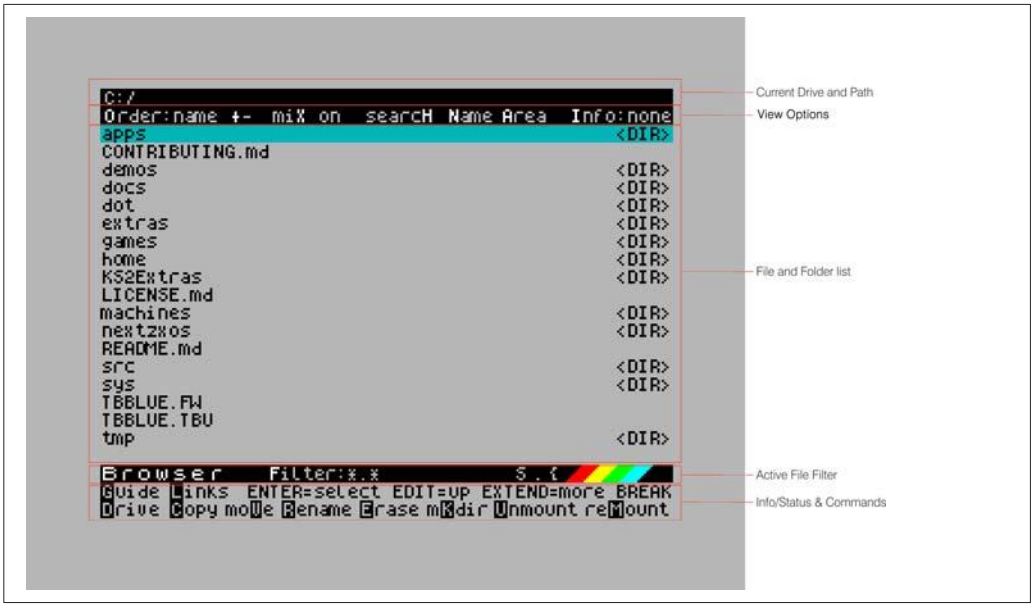



*Fig. 22 – Browser window areas and their function*

On the top of the *Browser* window, is the *Current Drive and Path Area*. As you navigate your drives, it changes to reflect the current drive and folder you're in. This in effect, is the same as giving the **PWD** command when in *NextBASIC*.

Right below that are the *View Options* which control the way the list will appear, from what info it will show to how the list is sorted together with a search control

Immediately after, is the *File and Folder List Area*; it contains all files and folders at the point you're located as reflected by the *Current Drive and Path Area* at the top in combination with the *Active File Filter Area* that's right below it (more about that in a little bit) shown in pages of 19 items at a time. You navigate the file and folder list with the cursor keys, **ENTER** and **EDIT**, a joystick set as cursor, or the first *Kempston* or *Mega drive* joystick regardless of what *port (*Left or Right) it's set to. Immediately below the *File and Folder List Area*, is the *Active File Filter Area* with which, you can reduce the file list to whatever *types* (including folders which have essentially a *blank type*) you wish to see (according to a filter set by *wildcards*) and finally, the bottom two lines is the *Info/Status and Commands Area*.

## Using the Browser

The Browser is extremely easy to use; all it takes is a few keystrokes to accomplish most tasks. Controls are listed in the next table:

| Key | Description |
|---|---|
| ⇦ | Moves one page up or to the topmost item if you're on the first page |
| ⇨ | Moves one page down or to the last item if you are on the last page |
| ⇧ | Move up one item |
| ⇩ | Move down one item |
| ENTER | If it's a folder, change to that folder. If it's a file attempt to execute it |
| SYMBOL SHIFT + ENTER | Attempt the secondary action stored in browser.cfg for the file type |
| EDIT | Move up one folder |

*Table 17 – Browser controls*

while commands are the following:

| Key | Description |
|---|---|
| O | Cyclically changes the sorting of the files between None, Name, Size and Date |
| + | Sorts the display Incrementally |
| - | Sorts the display Decrementally |
| X | Toggles whether Folders and Files will be mixed or separate |
| H | Performs a search for a specific file |
| N | Shows the full name of the currently selected object |
| A | Switches User Area |
| I | Toggles the Info display between None, Size, Date and Attributes |
| D | Cyclically changes the drive to the next in the list of mounted drives |
| K | Makes a new Folder |
| R | Renames the currently selected item |
| C | Selects the currently highlighted file for copying |
| E | Erases the currently selected item |
| M | Remounts all drives |
| U | Unmount current drive |

*Table 18 – Browser commands*

In order to *copy* a file, you will need to highlight (using the cursor keys) the file and then press **C**. The status lines will change to: **Copy? (Y/N)** to which you'll need to reply with a **Y** or **N** (for Yes or No). Then you navigate to the new location whether this is on the same drive or on another drive and once you've reached your intended target you will need to press **P**. The *Browser* will ask you if you want to **Paste here? (Y/N)** to which you'll need again to reply with a **Y** or **N**. If you attempt to copy a Folder (marked by a **<DIR>** on the file and folder list) the *Browser* will still ask: **Copy? (Y/N)** but it will silently reject any attempt to **P**(aste) the folder on another location.

*Erase* also asks a similar question; **Erase? (Y/N)** will appear after you highlight a file and press **E** but in the case of a folder it will fail with a **Dir Full** flashing error displayed in the *Status Area* if the folder contains any item in it.

*Rename*, as in the case for the **MOVE** command we examined previously, does three things: Renames and/or moves a file. You highlight an item and press **R**, and a **New name:** prompt appears in the *Status Area* asking you for a new name (or a new location together with the old or a completely new filename). Rename doesn't work across drives so no drive name is required in case of a move, which means that you can start the new name with a / or \ to indicate the root folder of the current drive. As a matter of fact entering any drive (even the current one) at the beginning of the new name *filespec*, will fail with a **No rename between drives** error.

You can *Rename/Move* a folder to be under another folder, however the latter must already exist otherwise Rename will fail with an **Invalid path** error.

To make a new folder/directory, the *Browser* has the *M(a)K(e) Dir* command, accessible by pressing **K** on your keyboard. The Status Area will change to display a **New name:** prompt. The new name must conform to the parameters of a folder *filespec* as discussed in the **MKDIR** command section earlier. As is the case with **MKDIR**, any attempt to create a folder in a drive that doesn't support it will result in a **Not implemented** error in the *Status Area*.

The *Browser* can *unmount* any drive *except* drive **c:** by switching to that drive using **D** and then pressing **U** on the keyboard and mount any *virtual disk image* it knows about (that is: **.dsk** and **.p3d** file types) by selecting it and pressing **ENTER**. It will ask you which drive letter you want to mount it on by displaying a **Mount on which drive? (A-P)** prompt followed by a **[A: is recommended]** in the case of +3 disk images. It will then display a **Try to boot disk now? (Y/N)** prompt. The latter process will try to load special files named ***** or **DISK** that exist inside the *disk image*. If auto booting is not possible a message: **Not bootable** will appear in the *Status Area*.

There is no way to remount a single, previously mounted physical drive that you chose to unmount through the *Browser*. You have, however the option to perform a complete *remount* operation by pressing **M** on your keyboard. Once you do that, you'll be prompted to

remove your SD card (this message applies to both SD cards) and once you press **Y** on the prompt, *NextZXOS* will perform the **REMOUNT** command as discussed earlier, thus remounting any physical drives you've unmounted.

## Configuring the Browser

File and drive management operations with the *Browser* is one facet of what it can do. The most important function it has however is to recognise and launch files of various types when we highlight them and press **ENTER** (or **SYMBOL SHIFT** + **ENTER** – *see immediately below*). It's able to do so due to its extensible nature using a simple, specially formed text file called **browser.cfg** that's located under **c:/nextzxos/**. The *Browser* also offers a way to assign TWO types of launching for a *filetype*. This is accomplished by adding two lines in **browser.cfg**. For example we could **LOAD** a **.bas** file or convert it to plain text using the **.bas2txt** *dot command*. The first action would be launched by **ENTER** and the second one with **SYMBOL SHIFT** + **ENTER**

Each line of **browser.cfg** contains information formed in the following fashion:

*TYPE LINE*

where *TYPE* is a 3 letter file type (e.g. **BAS**) followed by *LINE* which is a sequence of *NextBASIC* commands separated by colon characters as per usual but prefixed with one of the following symbols:

| Prefix | Meaning |
|---|---|
| : | Return to *Menu* afterwards |
| < | Return to *Browser* afterwards |
| ; | Return to *NextBASIC* afterwards |

The *NextBASIC* commands that follow, use the following placeholders:

| Character | Meaning |
|---|---|
| \| | Is replaced by the short *filename* as read by the *Browser*[18] |
| "\| | Is replaced by the long *filename* as read by the *Browser* and must be terminated by a matching quote (") |
| £ | Is replaced by *language code* (ie. **en** for English, **es** for Spanish etc) |

Additionally, if a quote character is needed inside the *NextBASIC* command sequence, it can be *escaped* using the backwards slash character as follows \".

*Wildcards* can be used to replace parts of a file *type* (***** for the remainder, **?** for only one character)

**Browser.cfg** can be edited using any standard text editor. More information about the Browser and how to configure it can be found by launching its *guide* file with:

```
.guide browser
```

## The Command Line

The *NextBASIC editor* is excellent for editing large programs, however for single use commands like the ones for file management or the *dot commands* we have been examining on a case-by-case basis, it can be a bit cumbersome to use, especially since the underlying *NextBASIC* listing will appear after every direct command. For that reason, *NextZXOS* includes a special version of the *NextBASIC editor*, that hides (but does not erase) any *NextBASIC* program that you may be editing and offers an uncluttered view of the screen making it easier to enter commands directly to the operating system. Unlike other operating systems, the *NextZXOS* command line still gives full access to *NextBASIC* and doesn't include a prompt like the one available on *CP/M* which we'll examine a bit further. To access the *Command Line* interface, press **EDIT** to bring up the *NextZXOS menu*, select *Command Line* and press **ENTER**. While in the *Command Line* interface you have the op-

18 *This functionality is recommended with dot commands that cannot deal with LFNs.*

tion to change how many columns are displayed by either again calling up the *NextZXOS menu* with **EDIT** and selecting the *32/64/85* entry or by directly giving the **SPECTRUM CHR$** command that can change the columns displayed immediately. See the **SPECTRUM CHR$** entry previously in this chapter for details of usage.

Notes



WARNING! WARNING! WARNING! WARNING! WARNING!

**Disabled Expansion Bus** refers to **disabled SIGNALS** on the Expansion Bus. The Expansion Bus is **CONSTANTLY UNDER POWER** and you must **ALWAYS PLUG Interfaces and ROM cartridges** with **ALL CABLES DISCONNECTED** otherwise **IRREPARABLE DAMAGE MAY OCCUR!!!!**

WARNING! WARNING! WARNING! WARNING! WARNING!

## ROM Cartridge Loaders

For users of ZX Interface 2, Ram Turbo, Dandanator and compatibles, *NextZXOS* introduces the ability to load ROM cartridge based software directly from the *More… submenu* and selecting the *Interface 2* option. Since the ZX Spectrum Next starts with the expansion bus disabled, it provides a quick way to type the appropriate commands to load either 48K or 128K ROM based software as well as apply all necessary settings to ensure maximum compatibility of cartridge based software. All you have to do is select the appropriate option. *NextZXOS*, will make the necessary adjustments, enable the bus and load the software.

## 48K BASIC

The *48K BASIC menu*, located in the *More… submenu*, turns your ZX Spectrum Next to into a standard 1982 ZX Spectrum… with a twist! First of all, according to the Next personality you have selected during boot, you may have full key entry (*Looking Glass*) instead of token (i.e. the keywords you see printed on your ZX Spectrum Next's keyboard) single-key entry (ZX Standard). Additionally, you have access to all the ZX Spectrum Next's additional features although not from BASIC. Finally you have access to your SD card via the *dot commands* we've already discussed. You can also reach 48K BASIC using the **SPECTRUM** command as discussed in a previous section.

## 128K BASIC

The *128K BASIC menu*, located in the *More… submenu*, turns your ZX Spectrum Next to into a 1985 ZX Spectrum 128K with the extra hardware of the Next available but unlike the 48K option discussed above, the dot commands do not work as esxDOS requires a 48K BASIC (the so called USR 0 mode).

## ZX80 and ZX81 BASIC

This is a convenient way to access the ZX80 and ZX81 emulators by Paul Farrow without having to launch a separate personality on boot. There's no way other than reset to come back from the ZX80/81 emulators.

## NMI Menu

While in Next mode, pressing the **NMI** button will launch the *NMI menu* which provides a lot of useful functionality to your ZX Spectrum Next. The *NMI menu* traces its lineage back to an expansion interface called *Multiface*. Multiface, allowed users to pause a program and *break into it*, create snapshots of the system's memory which upon reload, placed the machine in the same place they were (and running the specific program they were) at the point in time they were, when they saved each snapshot.

The *NextZXOS NMI menu* offers, however, many more features over those of the original Multiface. We'll examine the most important ones of these below.

Upon loading, we can see the following entries in the menu:




*Fig. 23 – NMI main menu*

*Return* – turns off the *NMI menu* and returns you to whatever you were doing prior to pressing the **NMI** button.

*Snapshot* – Produces a snapshot of any legacy software that's currently running. It automatically recognises if it's a 48K type or 128K type of software and adjusts the snapshot type produced accordingly.

*Screenshot* – Produces a screenshot of whatever is in any of the layers' screen memory areas and prints (to a ZX Printer or compatible) a ULA (*Layer 0*) screenshot. It also saves and restores the current palettes.

*TAP Files* – Manages the redirection of input and output to drive **t:** (tape) to virtual tape files (**.tap**) as well as browses their contents (In essence a shortcut to **.tapein**, **.tapeout** and **.lstap** we've covered previously).

*POKEs* – Manages and applies **.pok** files to running software. These are files containing known workarounds and patches to specific applications – used mostly for games; for infinite lives etc.

*Debug tools* – Gives access to maybe the most powerful set of features in the entire suite: A *Next Register* and *Z80n* Register status browser, a *memory map* and *bank browser*, the ability to *set breakpoints* in memory to intercept running code as well as a *banked memory save tool*.




*Fig. 24 – NMI Settings*

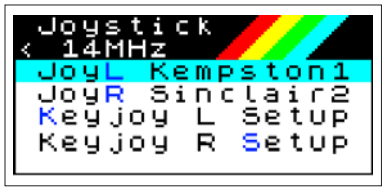



*Fig. 25 – NMI Joystick Settings*




*Fig. 27 - NMI General Settings*




*Fig.26 – NMI Sound Settings*

*Settings* – Allows easy modification of hardware settings on-the-fly, from the ones available on the configuration menu to the ones that are more nuanced (like the type of DMA chip in use or the machine timings used in the specific personality) which aren't always available through the standard configuration (*Fig. 24* through *27*).

Keymap – This is a duplication of the **.keyhelp** *dot command* and provides a quick on-screen legend of the keyboard tokens (for the 48K mode) which is particularly useful if using a board-only Next or a PS/2 keyboard.

*About* – Displays a *NextZXOS About* screen with several credits to contributors of bug reports and suggested features.

The *NMI menu,* uses the familiar *Browser* interface *dialogs* for loading and saving of files as needed as can be seen in the next figure.

## The NextZXOS folder structure

To achieve a *complete* and properly booting *NextZXOS* the following folders and files need to be present on an SD Card:

At the root level there's the *Firmware* file (**TBBLUE.FW**) and the folders **c:/nextzxos/** carrying all the drivers (*RTC*, *Mouse* etc), support programs and overlay files as well as the base *CP/M* image file together with the startup command file **autoexec.bas**.

Then there is **c:/machines/next/** which contains two versions of the *NextZXOS* ROM (they differ in the type of 48K ROM they contain; *Sinclair* or *Looking Glass*), the *NextZXOS divMMC* ROM, the *NMI* ROM as well as the configuration file **config.ini** which tells the *Firmware* the particular settings you require for your machine.

Finally, there is **c:/dot/** which contains, apart from the third party *dot commands*, the ones that constitute part of *NextZXOS,* namely: **.$**, **.bas2txt**, **.browse**, **.browseprefs**, **.cpm**, **.defrag**, **.editprefs, .guide**, **.install**, **.lfn**, .mem, **.mkdata**, **.mkswap**, **.nextver**, **.txt2bas, .uninstall** and **.unzip**.

To obtain just a booting *NextZXOS* you do not need the *dot commands*, *CP/M* base image, *mouse driver, RTC driver* or even **autoexec.bas** and *NMI* rom. Your functionality however will be limited.

## NextZXOS dot commands

We have talked about *dot commands*, covering each one as the case dictated, but we haven't talked about what they actually are! Well, dot commands are basically an easy way to add functionality to *NextBASIC* (and ZX BASIC) originally invented for use by *esxDOS* by its author, Miguel Guerreiro. Copying from the *z88dk*[19] documentation by Allen Albright: *A dot command is loaded into an 8 K ram page located at address 0x2000, overlapping the rom, and can run without disturbing the basic system. They are launched from basic by typing their names with a leading dot, hence the name "dot command". Any string following the dot command's name is passed as a command line. On return the dot command can generate esxdos errors in the basic system, either canned ones or custom ones*.

*NextZXOS* has extended the scheme while remaining compatible with the original specification, thus a separate set of *dot commands* is included with **System/Next™**, than what comes with *esxDOS*. See the *esxDOS* section below for more details on the differences.

The scope of this manual is a bit limited to cover *dot commands* in their entirety but you can visit: **https://github.com/z88dk/z88dk/tree/master/libsrc/_DEVELOPMENT/EXAMPLES/zxn/dot-command** to find out more about how they work and how you can write your own.

*NextZXOS,* apart from the third party ones included in the **System/Next™** distribution, has several *dot commands* that perform special functions not covered elsewhere. These are:

| | |
|---|---|
| **.$** | *Dot commands* cannot accept string arguments from *NextBASIC*, so **.$** allows execution of a *dot command* accepting any parameter passed as a string thus enabling full integration of *dot commands* in *NextBASIC* |
| **.bas2txt** and **.txt2bas** | *NextBASIC* is stored in a *tokenised* form. That means that each keyword occupies one token (see *Appendix A* for these values). |

19 *z88dk is a C-based cross-development system for a variety of Z80 compatible CPUs and systems.*

| | |
|---|---|
| | That further means, that it's only machine and not human-readable other than from within the *NextBASIC Editor*. These two *dot commands* allow *NextBASIC* to be exported to a text file to be edited by a more specialised programmer's editor, or shared with other, non Sinclair computers and imported back in a form that the NextBasic Editor can understand. |
| .browse | One of the nicest features of the *Browser* is its built-in file dialogs. **.browse** allows these to be used within your *NextBASIC* programs and pass the selected file to a string variable in your program saving immense amounts of time from programming menu-based navigation. |
| .defrag | *NextZXOS* provides a streaming API which can be used for audio or video. If the files however are not defragmented, streaming is interrupted. **.defrag** solves this problem rearranging the file in question to be in one, continuous, piece. |
| .editprefs /.browseprefs | NextZXOS's native customisers for the editor and the browser. |
| .guide | **.gde** is the official documentation NextZXOS file format and NextGuide is its viewer. It's a hypertext viewer partially compatible with the Amiga Guide Format |
| .install/.uninstall | These are the *dot commands* to *install* and *remove* drivers like for example the *mouse driver* from the system. *NextZXOS* provides a driver API, which you can use to write your own drivers which is used in conjunction with the new **DRIVER** command. |
| .lfn | This is a very special use case command; its sole purpose is to return the long file name for a short (8+3) filename. **.lfn** does not work on *IDEDOS/+3DOS* drives, or rather it does work but returns the same name as *+3DOS* drives only accept 8+3 filenames. |
| .mem | Returns the free memory for *NextZXOS* and *NextBASIC* use |
| .nextver | Assigns the current version of *NextZXOS* to a variable we specify. |
| .unzip | Native decompressor for **.zip archives** |

**Notes**

Any errors generated by a *NextZXOS* dot command generate an error code of **255** (**Dot Command Error**) which can be read with the **ERROR. ERROR$** and **ERROR TO** commands. Refer to *Chapter 1* for details.

## Modifying the startup – Autoexec.bas

*NextZXOS* provides you with a very fast way to set up your *NextBASIC* and *NextZXOS* environment upon boot by using commands stored in a special file called **autoexec.bas** located inside the **c:/nextzxos/** folder. The same rules apply as with regular **SAVE**, meaning you will need to give a **LINE** parameter to save it before it can auto execute. If you omit the **LINE** parameter, the commands will auto load upon boot but won't execute. For example to set up a red background with bright white letters upon boot:

```
10 SPECTRUM PAPER 2: SPECTRUM
   BRIGHT 1: SPECTRUM INK 7
20 ERASE: REM ERASES ALL LINES
```

Then

```
SAVE "c:/nextzxos/autoexec.bas" LINE 10
```

Reset and… magic!

## CP/M

The ZX Spectrum Next supports running *CP/M Plus* (also known as *CP/M 3.0*), an operating system available for many microcomputers in the late 1970s and early 1980s.

*CP/M* provides a command-line environment similar to *MS-DOS.* A huge amount of software was available for it, including programming languages, both interpreted and compiled, word processors (such as the well-known WordStar), spreadsheets, databases, utilities, text-based games and much more.

The ZX Spectrum Next runs *CP/M Plus* using a specially-written *BIOS* (Basic Input/Output System) which gives it a 80 x 24 text-based terminal supporting full colour.

To run *CP/M*, you need to call up the *NextZXOS Startup menu*, go to the *More… submenu* and select the *CP/M* option or from *NextBASIC* or the *Command Line*, use the d*ot command* .cpm.

Any software, compatible with *CP/M-80*, *CP/M 2.2*, *CP/M 3.0* or *CP/M Plus* will work on the




*Fig. 28 – Initial CP/M setup procedure*

ZX Spectrum Next's flavour of *CP/M* except *CP/M-86* software (which requires an Intel x86 processor) and *CP/M-68* software (which requires a Motorola MC68K class processor).

Please note that *CP/M* graphical applications requiring *GSX* cannot be used at the moment, although support for these is under consideration. This is not affecting software availability considerably, as there is very little software requiring *GSX*; most *CP/M* software was text-based.

## Getting started

Before you can use *CP/M*, *NextZXOS* will need to prepare it. This is a process that happens automatically just once. You will need to access your *NextZXOS Startup Menu*, then from the *More…* option, select the *CP/M submenu*. NextZXOS will start working on its own and once it finishes, it will exit back to *NextZXOS*. From then on, every time you choose the

*CP/M* option from the *More… submenu* in the *NextZXOS Startup Menu*, or type **.cpm** in the *NextBASIC Editor* or the *Command Line* will take you straight into *CP/M (Fig. 29).*

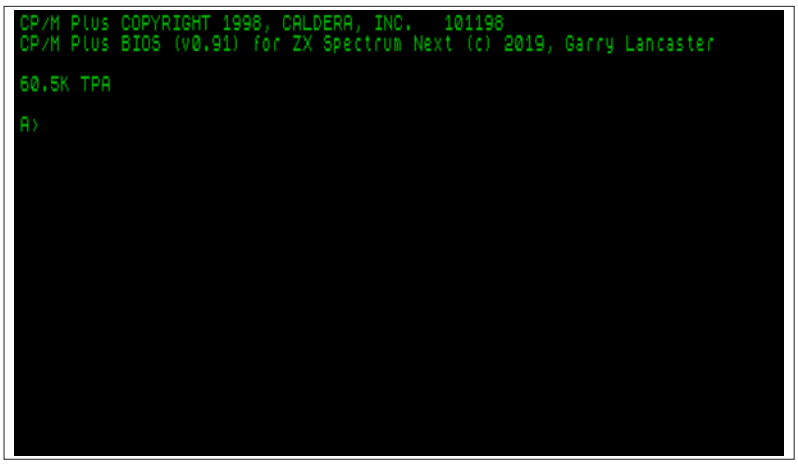



*Fig. 29 – ZX Spectrum Next properly booted CP/M setup*

## Commands

*CP/M* is operated by typing commands at the prompt (**A>**). One of the most useful commands is **DIR** which works much in the same way that **CAT** works in *NextZXOS*.

Typing:

**DIR A:**

will show a list of all the files on the current drive or the drive specified. Initially you will just have drive **A:** available, but more can be set up (drives **A:** to **P:** can be used) using the **.mkdata** *dot command* in *NextZXOS* as per the instructions provided earlier, so that you can keep different programs on different drives.

Any filename shown by **DIR** which ends in **.COM** is itself a command, and can be executed at the prompt. You will have noticed there are a lot of **.COM** files to try. Another useful one is:

**HELP.COM**

which provides help and information on all the standard commands and utilities provided with *CP/M*. Note, that you do not need to type the .COM part all the time; CP/M will find the appropriate command and executed without having to type its extension (in other words its file type). So to call up **HELP.COM** you could just type:

**HELP**

Commands are also case-insensitive, so it doesn't matter if you type them in lower or upper case or a mix of both; all versions of **HELP**, **help**, **hELP** and **HelP** will call the exact same program!

In the *CP/M* distribution that comes with *NextZXOS*, there are a number of commands specific to the ZX Spectrum Next. These include:

| Command | Description |
|---|---|
| UPGRADE | Upgrades your installation of CP/M from the latest version available on your SD card |
| TERMINFO | An interactive demonstration of the terminal facilities provided on the ZX Spectrum Next |
| EXIT | Exits from CP/M and returns to NextZXOS |

| Command | Description |
|---|---|
| COLOURS | Changes the colour scheme |
| TERMSIZE | Changes the default terminal size (up to 80 x 32) |
| IMPORT | Imports files from your NextZXOS c: drive (or other FAT drives seen in the NextZXOS browser) |
| EXPORT | Exports files to your NextZXOS c: drive (or other) |
| ECHO | Sends text or escape sequences to the terminal |
| NEXTREG | Views or changes ZX Spectrum Next hardware registers (use at your own risk!) |

Typing the name of these commands will give some more information on how to use them.

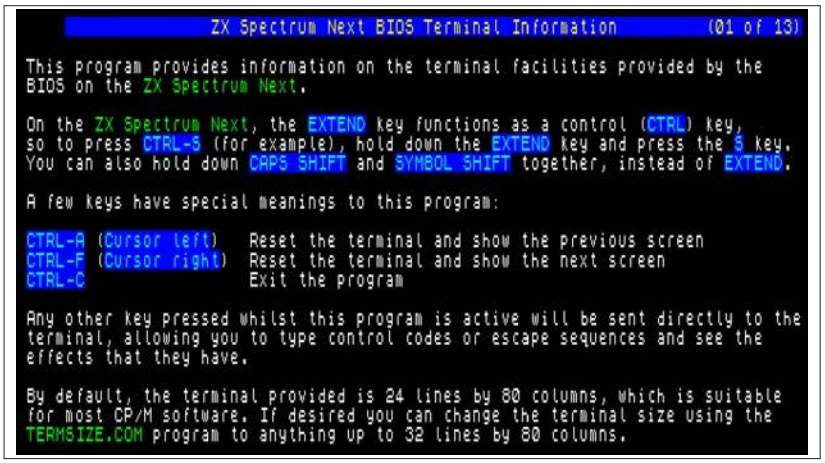



*Fig. 30 – TERMINFO output*

## Drives and CP/M

*CP/M* on the ZX Spectrum Next cannot access the standard SD card drive **c:** (or other drives you may have due to having additional SD cards inserted, for example). This is because *CP/M* directly accesses disks at a low level, and is incompatible with *FAT filesystems*.

Therefore, on the ZX Spectrum Next, *CP/M* uses *virtual disk* files. These can either be **.p3d** files (created by the **.mkdata** *dot command*) or **.dsk** files (images of standard ZX Spectrum +3 disks).

You can access multiple disk images at once in *CP/M*. To do this, simply create additional files with **.mkdata** using the same naming scheme. eg. at the *NextZXOS* command line, type the following:

```
.mkdata "/nextzxos/cpm-b.p3d"
.mkdata "/nextzxos/cpm-e.p3d"
```

When you next use *CP/M*, you will have drives **A:**, **B:** and **E:** available. Note that you can have a drive **C:** in *CP/M* if you wish, but this is not the same as the **c:** drive used in *NextZXOS*.

Up to **15** *virtual disk images* can be used at once by *CP/M*, and they can be mapped to any drive **A** to **P**, simply by naming the files in any of these ways:

c:/nextzxos/cpm-X.p3d
c:/nextzxos/drv-X.p3d
c:/nextzxos/cpm-X.dsk
c:/nextzxos/drv-X.dsk

where X is the drive letter, from A to P. If you have created multiple files referring to the same drive letter, *CP/M* will use the ones named **cpm-X** in preference to the ones named **drv-X.** It has no preference over **.p3d** or **.dsk**, so if there is a **cpm-b.p3d** and a **cpm-b.dsk**, then the first one in the directory will be used.

Note that *NextZXOS* will also automatically mount these drive images (except any image where **X** is **c**) when it starts up. You can view them in the *Browser* (press **D** to change drives) and copy files between them etc. NextZXOS will mount **drv-X** files in preference to **cpm-X** files. You can also manually mount other disk images which don't follow the automatically-mounted naming scheme. To do this, just press **ENTER** on the **.p3d** or **.dsk** file in the *Browser*.




*Fig. 31 – ZX Spectrum Next CP/M running WordStar 4*

## Further information

There is a lot to learn about *CP/M*, and a lot you can do with it. Some useful places for further information are listed below:

**http://www.cpm.z80.de/** Contains a lot of manuals, documentation and software.

In particular, the *CP/M 3 User Guide*, *Command Summary and Programmers' Manuals* can be found in the following locations:

| | |
|---|---|
| **http://www.cpm.z80.de/manuals/cpm3-usr.pdf** | User Guide |
| **http://www.cpm.z80.de/manuals/cpm3-cmd.pdf** | Command Summary |
| **http://www.cpm.z80.de/manuals/cpm3-pgr.pdf** | Programmer's Manual |

A good starting point is also:

**http://classiccmp.org/cpmarchives/** which links to many more useful sites, collections of software, manuals, magazines and much more.

## Preparing your ZX Spectrum Next for esxDOS

Other than *NextZXOS, CP/M* and *+3e/IDEDOS*, your ZX Spectrum Next supports natively one more Operating System called *esxDOS*. This is especially helpful when running Eastern European software as the preferred method of storage is using *TRDOS* which *esxDOS* supports natively. Unfortunately the copyright status of some parts of *esxDOS* prohibits its inclusion in the **System/Next**™ distribution, but that doesn't mean you cannot install it yourself. *esxDOS* can be invaluable for personalities other than the *Next Native* one as it provides older model personalities with an easy way of managing *FAT* formatted SD cards. As is the case with *NextZXOS*, it too uses *FAT* as the primary filesystem and thanks to *NextZXOS*' design, it can therefore co-exist on the same drive without clashes.

In order to install esxDOS you need to do a few things first:

- Go to **www.esxdos.org** and download either the latest version or the one whose rom comes with the System/Next distribution. For correct operation, the minimum supported version is 0.8.6 beta 4
- Using a PC, Mac or Linux machine, unzip the contents of the *esxDOS* distribution onto a drive, connect the **System/Next™** SD card onto the same computer and then do the following:
  - Copy the **BIN**, **SYS** and **TMP** folders into the **System/Next™** distribution's root folder
  - Copy the **ESXMMC.BIN** file from the *esxDOS* root to **c:/machines/next/**
  - Finally, edit the **config.ini** file in **c:/machines/next/** to include *esxDOS* with the personality you choose (Note that this doesn't apply to Next Native mode)

Here is an example that will modify config.ini to use esxDOS with the 128K personality (note that esxDOS will boot any 128K personality in what is called USR0 mode; a special mode where the editor is 48K but all the 128K features are available). After you download the esxDOS distribution archive from the esxDOS site, unpack it and follow the instructions above. Then go to **c:/machines/next/** and using any text editor (for example *Notepad* under Windows) open **config.ini**. Locate the line reading:

**menu=ZX Spectrum 128k,1,8,128.rom**

and modify it as follows:

**menu=ZX Spectrum 128k,1,8,128.rom, esxmmc.bin,<none>**

Also, if you have an *RTC* chip installed, go to **c:/nextzxos/** and copy **RTC.SYS** to **c:/sys/**. Save it, eject the SD card and transfer it to your ZX Spectrum Next. Upon boot, press **SPACE** and then using the cursor keys, locate the **ZX Spectrum 128k** line. Press **ENTER** and in a few seconds you'll see something like this:




*Fig. 32 – ZX Spectrum Next running esxDOS 0.8.6*

That was it, you now have a functioning *esxDOS* installation for your 128K personality on your ZX Spectrum Next computer and the green **Drive** button on the left side of your computer will start functioning calling the esxDOS browser.

# Chapter 20 – Channels, Streams, Drivers and Windows

As we have seen thus far, *NextBASIC* can *read data* from the keyboard and controllers using **INPUT** and **INKEY$** and it can *write data* onto the display or a printer by using **PRINT** and **LPRINT**. However, these commands are really a form of shorthand designed to protect the user from some of the computer's more complex features.

To the **PRINT** command, for example, there is no difference between the screen and the printer. **PRINT "Mikayla"** really means: *take the characters which make up the word **Mikayla** and send them somewhere else*. It's just convenient to use the screen most of the time. Likewise, **LPRINT** usually sends data to the printer. In fact, what these commands really do is to send data to one of a number of *channels*.

## Channels

A *channel* is the pathway to the computer's input and output devices and on the ZX Spectrum Next, they are designated by a letter. These are:

| Designator | Direction | Description | Default Streams | Default Status |
|---|---|---|---|---|
| k | Input/Output[1] | Keyboard | 0,1 | Open |
| s | Output | Screen | 2 | Open |
| p | Output | Printer | 3 | Open |
| i | Input | File (input) | | Closed |
| o | Output | File (output) | | Closed |
| u | Input/Output | File (Update) | | Closed |
| v | Input/Output | Variable | | Closed |
| m | Input/Output | Memory | | Closed |
| d | Depends | Driver | | Closed |
| w | Input/Output | Windows | | Closed |
| r | Internal | Internal Use only | N/A | Open |

*Table 19 – NextBASIC channels*

To access a *channel*, it must be open. Opening a *channel* makes it ready to receive or produce data. A *channel* is opened by connecting it to a *stream*. From *NextBASIC*, you would use a command like:

```
OPEN #4,"k"
```

which means *connect stream 4 to the keyboard channel*. As evidenced by the table above, if we go by the direction of data flow there are three types of *channels*: *Input*, *Output* and *Input/Output* (or *Update*).

However, we can better classify channels by device type: We have *Screen*, *Keyboard*, *Printer*, *File*, *Memory*, *Variable*, *Windows* and *Driver channels*. Let's examine them according to the device type however, as this affects what types of commands we can use with them and how.

The *Screen Channel* deals with everything that goes to the screen. It is the simplest of all channels and most of its characteristics have been covered in *Chapter 15* already. It is already opened and connected to *stream* **#2**. In fact you can substitute any **PRINT** command with **PRINT #2** and it will work in the exact same way as a regular **PRINT** command.

---

1 *Outputting data to the keyboard might seem a bit peculiar, but once you consider that the computer uses the lower screen (like **INPUT** does) to display the characters, it becomes clear why.*

Similar things apply to the *Keyboard Channel*. This is already connected to two *streams*: **#0** and **#1** as we can see by the following little program:

```
10 INPUT #0;"Stream 0 Input: ";a$
20 INPUT #1;"Stream 1 Input: ";b$
30 PRINT a$'b$
```

The *Printer Channel* is also simple and by default attached to *stream* **#3**. As a matter of fact, giving **PRINT** #3 is basically a default[2] longhand for **LPRINT** and similarly **LLIST** is basically the same as **LIST #3**.

**Notes**

As streams **#0** to **#3** are predefined and already opened, altering these may also alter the behaviour of the system, therefore you are advised to avoid the practice unless you exercise care.

Where things start to differentiate a bit is with the *Files Channel*. Firstly, no *file channel* is by default open, and secondly any file can be opened in 3 modes: *Input*, *Output* and *Update* (*Input*/*Output*). As the names imply, *Input* will only accept data FROM a file, *Output* will only direct data TO a file and *Update* will allow input and output of data TO and FROM a file. There are a couple of special considerations regarding *file channels:*

- You should always take care to close *streams* that have been opened to a file in *Output* or *Update* modes when you have finished, as otherwise data loss may occur. It is always good practice to do this even for files opened in *Input* mode (or *streams* open to other channels). The **CLOSE** command will be examined further below.
- Files saved by *CP/M* or a *+3e*, are usually stored as a number of **128-byte** *records* and so you may read rubbish at the end of a file that comes from such a system if it is not an exact multiple of **128 bytes** in length. *NextZXOS* however, reports proper file sizes and does not suffer from this problem even when it saves files on a +3DOS/IDEDOS drive.

*File channels* support all the *pointer commands* (more on these further below).

The *Variable Channels* can be used to direct output to or input from a string variable, which can be easily manipulated within a *NextBASIC* program. This would allow you to (for example) examine disk catalogues in your *NextBASIC* program, or make an auto-running game demo (by inputting from a string containing set keystrokes). The string specified must be a character array with a single dimension, large enough to hold the maximum amount of data you expect to have to deal with.

*Variable Channels* also support all the *pointer commands*.

The *Memory Channel* can be used in a very similar way to the *Variable Channels*. However, as it is a fixed memory region, it is more suitable for use by machine-code programs. It also requires you to reserve the memory beforehand.

The *Driver Channels* are special channels to exchange data with Device Drivers. Not every Device Driver can be addressed by a Driver Channel and not all Driver Channels have all options or can even access *pointer commands*. You will need to refer to each driver's documentation in order to know what is supported and what isn't.

Finally the most complicated *Channels* of all are the *Windows Channels*. Although they do not support any of the *pointer commands*, they are extremely flexible as they accept a large number of control codes as we've briefly mentioned in *Chapter 14*.

2 *Default means in this context: "without parameters". As we will see further below, even LPRINT and LLIST behaviour can change*

*Windows*, are defined by their top line (0-23), leftmost column (0-31), height (1-24), width (1-32), and optionally character size (3-8) and character set address. If no character size is specified, the default is 8. If a character set address is given, then this is used instead of the built-in fonts; this allows you to use nice fonts such as those provided with art programs and adventure games. Due to their complexity, we'll devote an entire section to *Windows* after we discuss streams and the commands with which we use them.

## Streams

*Streams*[3] are convenient ways for the computer to switch between channels by referring to them as numbers. This idea makes it possible to write programs that can send information to any device without having to use different commands. There are 16 total available streams numbered 0 to 15. 4 *streams;* 0 through 3*,* as seen on the table above, are already opened to channels k,s and p. Note here, that many *streams* can be attached to a channel depending on what we want to do.

## Using Streams

All the above might seem complicated, and you may well wish to stick to the standard PRINT and INPUT commands – that's why they're there after all. Even these commands however, are just shortcuts to their "complete" versions that also include a stream number and the benefits of using channels far outweigh their perceived complexity.

## Stream control commands

Since it's now evident that any device on the computer that accepts input or produces output is really a channel, it's easy to realise that we have been using streams all along; we've already visited PRINT and LPRINT (which are really the same command), used INPUT and INKEY$ and lastly, we've used LIST and LLIST (which also are the same command). All the above, have versions which include a # (hash) followed by a current stream number, so we are already halfway there!

Apart from these and OPEN # we saw in the channels section above, the following commands are available for working with streams: CLOSE #, DIM #..., DIM #...TO, NEXT #..., NEXT #...TO, POINT #..., RETURN #...TO, GOTO #...TO and COPY ...TO #, CAT # and PWD #. We'll examine them all below:

OPEN *#n, channelspec*

where n is the stream number[4] and *channelspec* is a string that can be any of the following (capitals or lower case letters may be used), opens a stream and attaches it to the channel defined by *channelspec*:

| String | Description |
|---|---|
| "k" | The standard input channel (keyboard and lower screen). Streams 0 & 1 are normally set to this channel |
| "s" | The standard output channel (main screen). Stream 2 is normally set to this channel. |
| "p" | The standard printer channel (serial or parallel). Stream 3 is normally set to this channel. |
| "i>*filespec*" | This opens an input-only stream to an existing file. If the filename is at least two characters long, you can omit the "I>" as this will be assumed (single-character names require the "I>" as otherwise they will be assumed to be standard channel names). |
| "o>*filespec*" | This creates a new file and opens an output-only stream to it. |

3 *On other versions of BASIC, streams are called channels and channels are called devices. This may be a bit confusing to a user coming from a different flavour of BASIC. The concepts however are basically the same.*

4 *Altering streams 0 to 3 will change the behaviour of the system and should be used with care.*

| String | Description |
|---|---|
| "u>*filespec*" | This opens an existing file and opens an input/output-stream to it. |
| "m>*address, length*" | This opens an input/output channel to the memory area at *address*, *length*. |
| "v>*x$*" | This opens an input/output channel to the variable x$ which must be a character array with a single dimension, large enough to hold everything that will be output to it/input from it. |
| "w>*line, col, ht, wid [, csize [, cset]]*" | This opens an input-output channel to a text *window* on the screen, starting at character position (*line*,*col*), with a height of *ht* character rows and a width of *wid* characters. Optionally, a character width of *csize* (3-8px) may be specified. This does not affect the definition details of the window, which are always specified in 8px wide characters. A user-supplied character set may also be specified, located at address *cset*. See the Windows special section for details. |
| "d>*driver_name*>*[driverspec]*" | Opens a channel to *driver_name,* whose data flow direction is dictated by the driver it addresses. *Driverspec* is optional and depends on the driver (if needed or not). |

*Table 20 – OPEN # channelspec setup strings*

Here are some examples:

| | |
|---|---|
| OPEN #4,"o>a:test.txt" | Creates a file named **test.txt** on virtual disk drive **a:** and opens an output-only channel to it, connected to stream **4**. |
| OPEN #5,"stuff" | Opens an existing file named **stuff** on the default drive and opens an input-only channel to it, connected to stream **5**. |

Once a stream is opened, it can be used with the standard **INPUT #** and **PRINT #** commands, as well as the additional *pointer commands*. Before we get into those, we should just first mention:

**CLOSE** *#n*

which closes the previously opened stream *#n*. If *n* is a stream between **0** and **3**, then the default channel for that stream (**k**, **s** or **p**) is reattached to it. Note, that attempting to **CLOSE** a stream that hasn't been opened, will not produce an error; instead it will exit gracefully with **OK, 0:1.** For example:

| | |
|---|---|
| CLOSE #4 | Closes the channel attached to stream **4**. |

Streams, and especially those opened to large files, can be very long to navigate in a serial manner; imagine having a file that's 100 Kbytes long, you would have to iterate through 102400 characters to read the very last one byte. For that reason, *NextBASIC* maintains *pointers* to the position you're located within a stream, how long the stream is (in characters / bytes), the ability to move these *pointers* to any location within a stream and finally the ability to read one byte from the current pointer position from that stream. The commands and functions to do that are called *Pointer Commands* and are the following: **POINT #...** and **RETURN #...TO**, **DIM#...** and **DIM #...TO**, **GO TO #** and **NEXT #...TO**. Let's visit their syntax below:

**POINT** *#n*
**RETURN** *#n* **TO** [%]*var*

This command returns the current position of stream *n.* It's the same as the **RETURN #... TO** with the exception that no variable assignment is done to the resulting value. If the **RETURN** variant is used, then it also stores it in variable *var*. The variable can be an integer one, which means that it will accept *–safely–* positions of up to **65536** bytes within the stream (or a maximum value of **65535** as position **0** is the very first position within a stream). Do not use integer values if you plan on accessing streams larger than that! If you don't use the **TO** variant however you can use it as part of the regular expression evaluator.

DIM #*n*
DIM #*n* TO [%]*var*

This command returns the size (in characters or bytes) of stream *n*. Whatever applies to the RETURN #...TO variant of POINT # applies to the DIM #...TO as well. Variable *var* stores the size of the stream. As with RETURN #...TO above, *var* can be an integer variable in which case the same warning as with the previous section applies.

GO TO #*n*, [%]*pos*

This command sets the current position of stream *n* to position *pos*. Let's see how the previous three commands all tie together by experimenting with browser.cfg:

```
10 OPEN #4,"/nextzxos/browser
   .cfg"
20 REM "i>" is optional since
   the filename is longer
   than 1 character
30 DIM #4 TO %a: REM Get
   filesize and put it in %a
40 RETURN #4 TO %b: REM Get
   current location and put
   it in %b
50 PRINT "You're in byte: ";
   %b ; " of "; %a
60 GO TO #4, %a/2: REM Move
   to the middle of the file
70 RETURN #4 TO %b: REM Get
   current location and put
   it in %b
80 PRINT "Now, you're in
   byte: "; %b ; " of "; %a
90 CLOSE #4
```

NEXT #*n* TO [%]*var*

This command gets the next character of input from stream *n.* As with POINT # and DIM #, if used with the TO modifier it also stores it in the variable *var*. If used on the standard k channel, this is similar to the INKEY$ function, except that it always waits for the next character to become available (ie on the k channel, it waits for a keypress). Using an integer variable here, is safe as the command gets one character at a time ergo one byte so its value will never exceed 255.

You can use this command instead of INPUT # on all channels that accept input otherwise they're very much identical in function.

Try this little program which will turn your ZX Spectrum Next into a typewriter:

```
10 NEXT #0 TO x
20 PRINT CHR$(x);
30 GO TO 10
```

Alternatively you could change line 10 to:

```
10 x=NEXT #0
```

which does the same thing!

COPY *filespec* TO *#n*

We've seen this command sequence before in a *shortcut* which did not include a stream number but rather a keyword: SCREEN$. In that case *n* is the stream to channel **s** which by default is #2. When used with a stream number, COPY...TO #n, can be used to transfer the contents of a file to a stream. For example to write the extended version of COPY "c:/readme.md" TO SCREEN$ we should type:

```
COPY "c:/readme.md" TO #2
```

When *NextBASIC* is running, it has four streams normally open. Streams #0 and #1 are connected to the keyboard (channel **k**), and are used by INPUT and INKEY$. Stream #2 is connected to the screen (channel **s**), and is used by PRINT, LIST, CAT and PWD, commands in other words that print something to the screen. Stream #3 is connected to the printer (channel **p**), and is used by LPRINT, LLIST and COPY (without parameters). All of these commands can be redirected to use another device by including a # followed by an open stream number, so

```
PRINT #1;"This is the lower screen"
```

will print the message on the lower screen while

```
PRINT #3;"Who needs LPRINT, Romulus?"
```

will use the printer. Conversely, LPRINT can behave like PRINT and typing:

```
LPRINT #2;"Are you confused yet Roy?"
```

makes LPRINT #2 do what PRINT normally does.

**Notes**

INPUT # may be used with other channels other than **k** and **w** such as file(**i**,**o**,**u**), memory (**m**) and variable (**v**) channels. In these cases, it is advisable to avoid any accidental outputs to the channels, by not using any prompt strings, and by using only the semicolon as a separator. In most cases, you will want to input a string using the LINE (See *Chapter 14*) modifier as without this, the data in the file (or other channel) would need to be surrounded by quotes.

## The Variable and Memory Channels

In the previous chapter, we've examined a special dot command (.$) that allowed *NextBASIC* to *talk* to any dot command not made specifically to interact with it. The Variable and Memory Channels can be seen as facilitating the reverse flow of information; to get information from the outside world into *NextBASIC*. They both involve reserving some space beforehand to accept the input but they differ in the sense that the former can be moved anywhere in memory (as variables could be stored anywhere) while the latter is a fixed location (which makes it more suitable for use by machine code programs). You may remember the series of commands we used to get the output of PWD in *Chapter 19* or .time in *Chapter 17*. Let's remember them quickly:

```
DIM d$(255):OPEN #2,"v>d$":.cd --verbose:
CLOSE #2: PRINT d$
```

and

```
DIM t$(100):OPEN #2,"v>t$":.TIME :CLOSE
#2:PRINT t$
```

but now that you know a bit more about streams, should that even work? The answer is yes, as it's designed to work that way. Most dot commands that produce textual output in a "legal" way (that is without circumventing *NextZXOS*), will attempt to output content on stream #2. By opening stream #2 to the variable channel and then executing the command whose output we wish to capture, we're performing a temporary redirection of the screen stream to the variable channel. Then, once we close the stream again, as the system is designed to do, it resets it to its default channel **s** and reopens it. Obviously if a program does not use the inbuilt *NextZXOS* and *NextBASIC* routines to produce output, this will produce nothing. The example below, shows a more "traditional" way of using the variable channel by using the inbuilt facility of a command (CAT ASN in this case) to output to a different channel:

```
10 DIM a$(1000)
20 OPEN #8, "v>a$"
30 CAT #8 ASN
40 RETURN #8 TO l
50 PRINT "Assignment length
   is:";l;" chars"
60 PRINT "List is:"
70 PRINT a$( TO l)
80 CLOSE #8
```

**Notes**

If a stream operation fails (like in the example above), the stream will not automatically close. It is therefore a good practice to start all your programs that operate on a stream with a CLOSE # prior to actually performing an OPEN # operation for the first time. It's also even better programming practice to include ON ERROR error-trapping, on every stream operation (especially the ones that operate on File Channels) as a lot of things can go wrong while working with files and channels in general (e.g. Running out of data, or your reserved memory area was smaller than the one you should have reserved etc).

As you can see line 40 also demonstrates the use of a pointer command in the variable channel. If you do not reserve enough room (for the sake of displaying the results, change the size of a$ to just 10 characters from the 1000 it has) you will receive an 8 End of File error at line 30.

The memory channel operates in a very similar manner; once you reserve the space, you open it and dump the output to it. Let's modify the above program to use the memory channel:

```
10 CLEAR 29999
20 OPEN #8, "m>30000,1000"
30 CAT #8 ASN
40 RETURN #8 TO l
50 PRINT "Assignment length
   is:";l;" chars"
60 REM perform some magic
   here via MC
70 FOR f = 0 TO l-1
```

```
80 PRINT CHR$(PEEK(30000+f));
   :REM print the l first
   bytes you stored in memory
90 NEXT f
80 CLOSE # 8
```

## Installable device drivers and Driver Channels

As mentioned in the previous chapter, *NextZXOS* allows for installable device drivers. A maximum of 4[5] of those can be installed.

These are mainly intended for use as software that allows access to external or internal peripherals such as printers, mice, network devices etc, but can also be used for other purposes, such as a potential NUL driver which does nothing. (The notion of a device that does nothing is a bit peculiar but it has its uses in computing!). As mentioned in *Chapter 19*, to install or uninstall a driver, you need to use the following dot commands respectively:

.install *drivername*
.uninstall *drivername*

where *drivername* is the name of the file which contains the code for each driver. For example the WiFi driver for the ESP chip that your ZX Spectrum Next may have come with or you may have installed yourself is espat.drv.

The documentation that comes with the driver will describe how to use it. Some drivers for example may make use of the new DRIVER command. This has the following form:

DRIVER *driverid, callid [,n1[,n2]] [*TO *var1[,var2[,var3]]]*

where *n1* and *n2* are optional values to pass to the driver, and *var1*, *var2* and *var3* are optional variables to receive results from the driver call. The individual DRIVER commands that you can use, depend on each device driver and they will also be in the driver's accompanying documentation.

## Driver Channel support

Some drivers can support input/output via streams and the Driver Channel d. If so, the documentation will describe the exact format it supports. Generally speaking however, in order to open a stream to channel d, you will be using one of the following command variants (assuming the driver id is ASCII X):

```
OPEN #8,"d>X"
```

which opens stream #8 to simple driver channel for device X.

```
OPEN #8,"d>X>string"
```

which opens stream #8 to channel d as described by string on device X.

```
OPEN #8,"d>X,p1"
```

which opens stream #8 to channel d as described by numeric value p1 on device X.

```
OPEN #8,"d>X,p1,p2"
```

which opens stream #8 to channel d as described by numeric values p1 and p2 on device X.

5 *This number may change in subsequent versions of NextZXOS*

To close the driver's stream, you will use a standard **CLOSE #** command (in the examples above that would be **CLOSE #8**).

Once the driver's channel is open, you can use any of *NextBASIC*'s stream input, output or pointer manipulation commands (if these are supported by the loaded driver; Usually each driver's documentation should describe what can be used).

A good example of using the driver channels can be found in the documentation for the ESP (WiFi) driver by Tim Gilberts, included in the **c:/docs/extra-hw/** folder of the **System/Next™** distribution. You can see there for example that talking to the internet via *NextBASIC* can be as simple as:

```
OPEN #4,"d>N>TCP,145.239.200.34:80"
```

which will open a TCP connection to port **80** on **specnext.dev**

## Windows

*NextBASIC* offers the ability to create and manipulate text "windows" on screen via its Window Channels. This allows for immense flexibility in manipulating textual output, going beyond what simple **PRINT** commands can.

## System Windows vs User Windows

When we talk about *Windows*, we're really talking about two kinds; *System* and *User Windows*. The former are created and managed by *NextBASIC* while the latter are created and controlled by the user. By default, 4 *System Windows* are created; one for each Layer other than 0. These are full screen and are used to produce output through the standard **s** channel and only a few parameters of these can change (size always remains the maximum possible).




*Fig. 33 – NextBASIC Text Windows*

*User Windows* on the other hand can have varying sizes and can be defined anywhere in the screen. From now on, we'll refer to System Windows as *SW* and to User Windows as *UW*. If no designation exists, then the discussion applies to both types.

## Defining User Windows

User windows are defined by their top line (**0** to **23**), leftmost column (**0** to **31**), height (**1** to **24**), width (**1** to **32**), and optionally by character size (**3** to **8**) and character set memory ad-

6 *Memory address refers to an address location within the main memory map.*

dress[7]. If no character size is specified, the default is assumed which is 8 px wide. If a character set address is given, then this is used instead of the built-in fonts[8]; this allows you to use nice fonts such as those provided with art programs and adventure games.

The character size, has no bearing on the way the window is defined, but it does affect the number of actual columns you have available. For example, the following defines a window the size of the entire screen; but because a character size of **5** is specified, the number of characters that can be printed in the window at any time is 24 x 51:

```
OPEN #5,"w>0,0,24,32,5"
```

When outputting via **PRINT** to windows, you can use many of the same control functions as you can with the normal screen. For example: ' (apostrophe); start a new line, , (comma); start a new column, **TAB**, **AT**, **POINT**, **INK**, **PAPER**, **FLASH**, **BRIGHT**, **INVERSE**, **OVER**.

When first defined, windows are in *non-justified* mode, but they can be set to be *left*, *full* or *centre* justified. Note that in *justified mode*, some features and control codes cannot be accessed, so you may need to switch back to *non-justified* mode to use them.

A complete list of control codes follows in the table below; these codes can be sent to a window using **PRINT** followed by the **CHR$** function as we've already seen in *Chapter 14. Note that it's always preferred to use standard* **PRINT**, **AT**, **INK** etc commands instead of control codes when using windows as they're usually easier to use than their control codes counterparts. Below is a list of all control codes that can be used while outputting to a Window Channel's stream.

NOTE that wherever there are sequential numbers they must be given using semicolon separated CHR$ statements. For example:

```
PRINT CHR$ 29; CHR$ 2;
```

| J | Code | Description | |
|---|---|---|---|
| | | UW | SW |
| | 0 | Turn justification off | Increases the current character set width (can range from **3** to **8** pixels), and moves the cursor to the start of the next line. |
| | 1 | Turn justification on | Decreases the current character set width (can range from **3** to **8** pixels), and moves the cursor to the start of the next line. |
| | 2 | Save current window contents | Causes the size **8** character set to be replaced with the character set defined by the CHARS system variable. |
| | 3 | Restore saved window contents | Causes the sizes **3** to **7** character sets to be regenerated |
| | 4 | Home cursor to top left | |
| | 5 | Home cursor to bottom left | |
| X | 6 | Tab to left or centre of window (**PRINT** ,) | |
| | 7 | Scroll window | |
| X | 8 | Move cursor left | |
| X | 9 | Move cursor right | |
| | 10 | Move cursor down | |
| | 11 | Move cursor up | |
| X | 12 | Delete character to left of cursor | |

8 *A font is a collection of a stylised graphical representation of characters . For the ZX Spectrum Next, this follows the 8x8 pixel matrix of the UDGs and it is exactly 768 bytes long (defining 96 characters in the 7-bit Sinclair ASCII series from 32 to 128). See Appendix A for a list of characters.*

| J | Code | Description | |
|---|---|---|---|
| | | UW | SW |
| | 13 | Start new line (PRINT ') | |
| | 14 | Clear window to current attributes | |
| | 15 | Wash window with current attributes[8] | |
| ● | 16; n | Set INK n (where n=0 to 7) | |
| ● | 17; n | Set PAPER n (where n=0 to 7) | |
| ● | 18; n | Set FLASH n (where n=0 or 1)[9] | |
| ● | 19; n | Set BRIGHT n (where n=0 or 1)[9] | |
| ● | 20; n | Set INVERSE n (where n=0 or 1) | |
| ● | 21; n | Set OVER n (where n=0 or 1) | |
| × | 22; y; x | Sets cursor to pixel line *y*, character size column *x*. (AT y,x). Position is specified in terms of character positions (dependent upon the character size currently selected and whether reduced-height text is in operation. Double-width and double-height do not affect the coordinates, however) | |
| × | 23; nLow; nHigh | TAB to (character sized) column *n*. This is a 16bit number so for column numbers smaller than 256, *nHigh* is always 0. Otherwise *n* is calculated as nLow+(nHigh*256) | |
| ● | 24; n | Sets ATTR n (Where n=0 to 255)[10] | |
| × | 25; y; xLow; xHigh | Changes the print position to pixel coordinates *x*, *y* (0 to 511 and 0 to 191 respectively). Since we may be running at *Layer 1,2* mode (HiRes) and the *x* position may be higher than 256 pixels (ergo a value larger than what a single byte can hold) it breaks the *x* coordinate into two byte components: *xLow* (0 to 255) and *xHigh* (0 to 1). For horizontal resolutions up to 256 pixels, *xHigh* is always 0 while for resolutions > 256 pixels it may be 0 or 1. The *x* coordinate is calculated as (xLow)+(xHigh*256) | |
| ● | 26; n | Auto-pauses every *n* character lines. After each *n* character lines have been scrolled out of the window, output will automatically pause until the SPACE key is pressed (the bottom right character in the window will be flashed to indicate SPACE is being waited for).<br>After a window has been cleared, the first pause occurs before any lines have been scrolled out; subsequent pauses wait for *n* character lines. Typically you would want to set *n* to the height of the window. If set to 0 (the default), auto-pause is disabled. | |
| ● | 27; n | Fills window with character *n*. Attributes and cursor position are affected. | |
| × | 28; n | Sets double width (where *n*=1) or normal width (where *n*=0) | |
| ● | 29; n | Sets height *n* (0=normal, 1=double, 2=reduced, 3=double reduced) – See *Chapter 14* for details | |
| | 30; n | Selects justification mode *n* where *n* is 0=Left Justified , 1= Fully Justified and 2 =Centre Justified | Changes the current character set width to *n* (can be 3,4,5,6,7 or 8 pixels), and moves the cursor to the start of the next line. |
| | 31; n | Selects whether embedded codes are permitted (1) or not (0) in justify mode | Causes the size *n* character set to be replaced with the character set defined by the CHARS system variable. |

*Table 21– Window control codes*

In the table above on the column marked as J an × means *ignored if issued in justified mode* and an ● means *code can be used in justified mode only if the "embedded codes" setting has been enabled*. For control codes normally ignored in justified mode, note that these will still be taken into account if you set them before entering justified mode.

## User character sets

If the default character set(s) are replaced using control codes 2, 3 or 31 in a system window, any subsequent text printed in any window (which doesn't have its own user-defined character set) will use the new character set(s).

The system-defined character sets are partially shared: sizes 3 and 4 use the same set (only the leftmost 3 pixels are used for size 3), and similarly so do sizes 5 and 6. This should be borne in mind when replacing system character sets using control code 31.

8 *Has no effect on Layer 2 or LoRes*
9 *Ignored unless in Standard or HiColour modes and EnhancedULA is not enabled*
10 *Ignored in LoRes, Layer 2 and HiRes modes*

## Window input

Text windows support the INPUT command. If you use INPUT #, then a cursor is added to the window at the current position. You can then input any text desired, using the left and right arrows to move along the text input so far, or the up and down arrows to move to the start or end of the text.

The DELETE key deletes the character to the left of the cursor, and the ENTER key completes the input. Up to 191 characters can be accepted into each input variable.

## Window definitions

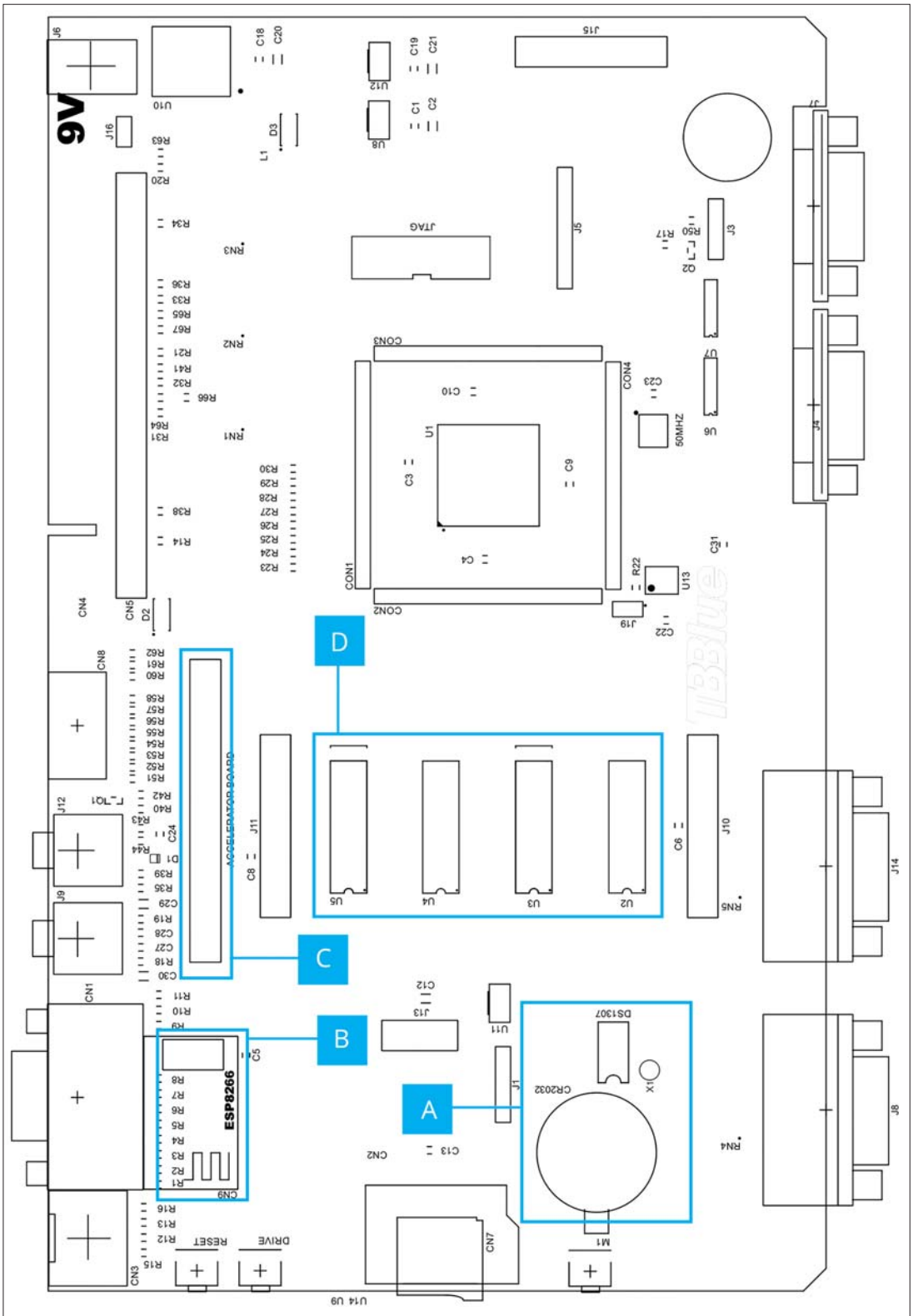
Since windows are defined using character squares so for example in LoRes, this means the maximum window size is 16 × 12 (and not 32 × 24). In HiRes however, character squares are considered to be 16 pixels wide, so the maximum window size is still 32 × 24 pixels.

## Memory constraints

It should be noted that saving/loading window contents (only available on user windows) is a costly operation. The amount of memory required for each character square is:

- 9 bytes (Layer 0)
- 16 bytes (Layer 1 HiRes or HiColour)
- 64 bytes (Layer 1 LoRes or Layer 2)

For example, a 10 x 10 window in Layer 2 requires 6400 bytes of available memory for saving its contents.




## Diagram Legend

**WARNING! WARNING! WARNING! WARNING!**
Before attempting any hardware addition, make sure all power is disconnected first!!!
ALL USER APPLIED MODIFICATIONS COME AT THE USER'S OWN RISK.
!!!IRREPARABLE DAMAGE MAY OCCUR!!!

| # | Description |
|---|---|
| A | Real Time Clock |
| B | WiFi module (ESP) |
| C | RPi0 Accelerator |
| D | Memory |

*The ZX Spectrum Next* ***Issue 2*** *Mainboard with optional equipment locations*

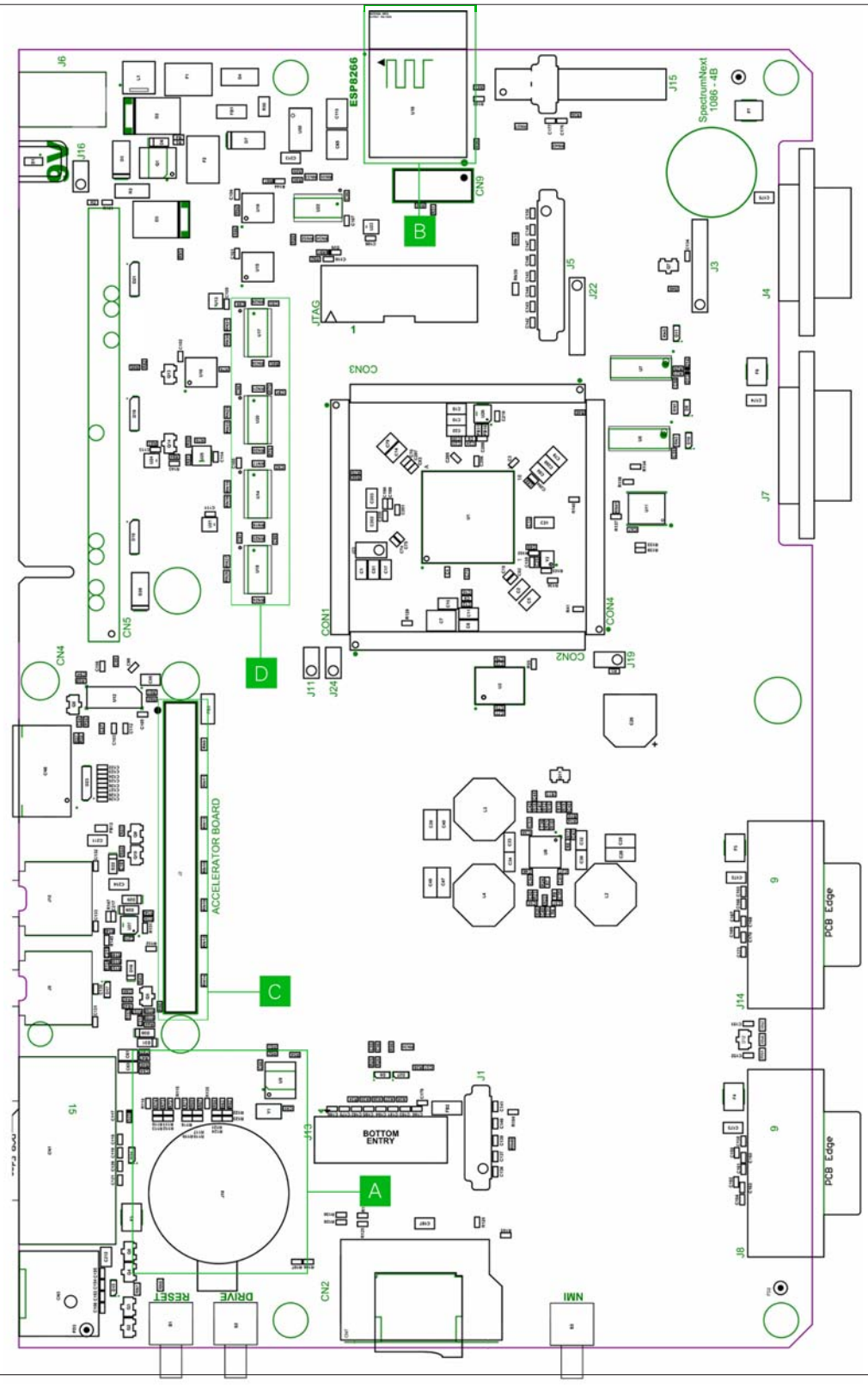



Diagram Legend

WARNING! WARNING! WARNING! WARNING!
Before attempting any hardware addition, make sure all power is disconnected first!!!
ALL USER APPLIED MODIFICATIONS COME AT THE USER'S OWN RISK.
!!!IRREPARABLE DAMAGE MAY OCCUR!!!

| # | Description |
|---|---|
| A | Real Time Clock (installed) |
| B | WiFi module (ESP) (installed) |
| C | RPi0 Accelerator (optional) |
| D | Memory (installed) |

*The ZX Spectrum Next* ***Issue 4*** *Mainboard with equipment locations*

# Chapter 21 – Optional Features

## Overview

Depending on the model you have and the Kickstarter you participated in, your ZX Spectrum Next may have one of two types of mainboards; Issue 2 or Issue 4 (See appropriate figures in the beginning of this Chapter). If you have an Issue 2, the rest of the chapter is for you, as many of the standard features on an Issue 4, used to be optional on the Issue 2. These are: *RTC* hardware, a WiFi module (*ESP*), extra RAM and the *Raspberry Pi Zero* (*RPi0*) accelerator.

If on the other hand, you have an Issue 4, the only feature you may be interested in is the Raspberry Pi Zero accelerator. The following sections will describe how to install and use them. Remember that modifying your ZX Spectrum Next carries a number of risks and that if you are not careful, you can damage your machine!

**Notes**



Before you proceed, please make sure that:

**A.** You're comfortable with tools and at least somewhat dexterous with a soldering iron
**B.** You understand the dangers of Electrostatic Discharge and sensitive components
**C.** You have enough patience –and finally–
**D.** You have downloaded and installed the latest System/Next™ distribution

## Installation (for Issue 2 mainboards)

Most add-ons are very easy to install with the exception of the *Real Time Clock* module and *RPi0 accelerator*. Installation of the former, requires soldering a number of parts onto the board and should be undertaken only by users with soldering experience. We recommend using a specialised service, if you do not feel comfortable with a soldering iron. Installation of the latter also requires soldering experience but that's confined on the RPi0 board itself and not on the ZX Spectrum Next. On the table below, we list all parts that you will need to perform each upgrade:

| Option | Parts Needed | Notes |
|---|---|---|
| 1024K Memory upgrade | 2 x Alliance AS7C34096A-10JCN –or–<br>2 x Samsung K6R4008V1D-JI10 | Upgrades the memory to 2048K |
| RTC module | 1 x DS1307 IC<br>1 x YXC YT-38, 32.768KHZ, 12pF oscillator or similar<br>1 x CR2032 Battery holder<br>1 x CR2032 Battery 3.3V<br>1 x 8 pin DIL socket (optional) | Allows time and date keeping that does not rely on your computer being powered on |
| WiFi module | ESP8266 ESP-01 | Provides access to the internet and your home network |
| RPi Accelerator | 1 x Raspberry Pi Zero<br>1 x Female IDC connector 2 x 20 pins | Various functions such as enhanced audio |

Installing a WiFi module, only requires you to populate the empty socket marked by a **B** on the diagram (page 218) by plugging in the *ESP* module in the place reserved.

Memory is equally simple, however, care must be exercised in that the RAM sockets accept larger chips than the ones the ZX Spectrum Next has. You need to line up the orientation notch (**B**) of each RAM chip (**A**) with the corner of the socket (**D**) leaving space (**C**) in the back of the socket. Once you have everything lined up, push with your finger at the centre of the RAM chip and it should make a slight click. While pushing the RAM in (and

every other module) make sure you provide enough support on the obverse so the board doesn't flex. Refer to the figure below on the proper installation of each RAM chip.

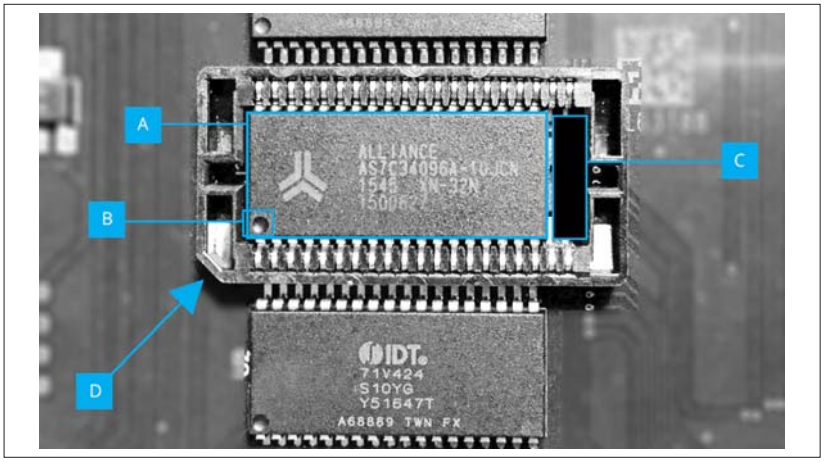



*Fig. 34 – Optional RAM upgrade installed*

The *Raspberry Pi Zero* (*RPi0*) accelerator requires a little bit of work. You will need to solder the 40 pin (2 x 20) FEMALE IDC header on the *RPi0*'s GPIO through-holes. Unlike what would be normally expected the socket needs to be soldered from the component side, therefore facing downwards. With a properly soldered IDC header you need to be able to see the *RPi0*'s SD card reader and all its components with the IDC header out of view like in the figure below:

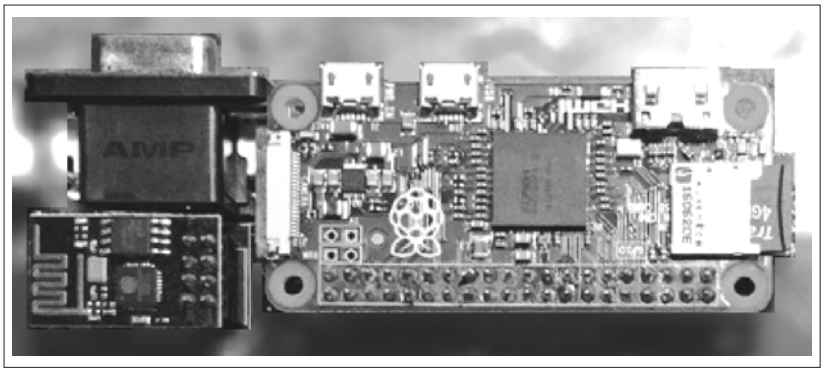


*Fig. 35 – Raspberry Pi 0 installed on Issue 2 board (with WiFi module in view - left)*

Please note that there is a backplate inside the case covering the holes of where the RPi0's USB, Power and Video out will have to appear from. Remove its screws and then pry it out gently with a flat screwdriver before attempting reassembly. Also pay attention to the Quick Start note regarding what is allowed to be plugged in the RPi0!!! This is also an ideal time to remove the expansion port backplate/cover if you plan on using your ZX Spectrum Next with external interfaces. *Figure 36* shows how a ZX Spectrum Next looks disassembled and there's special mention of both backplates.

The most complicated installation is that of the *RTC* module. It requires you to solder the oscillator in the X1 location of the board, a battery holder in the location marked and finally the DS1307 IC in its place next to the battery holder paying attention to the orientation (marked by a notch on the sketch on the board as well as on the chip itself). It's advisable that you install a 8 pin DIL socket instead of the DS1307 IC as heat may damage it during soldering.

You should exercise caution while soldering the oscillator; the through holes are very small and need to be free of any flux or solder residue as this will stop the oscillator from working. Finally, you will need to install the battery in the socket otherwise the *RTC* will only work for as long as the machine is powered.

## Raspberry Pi Zero installation on the Issue 4 mainboard

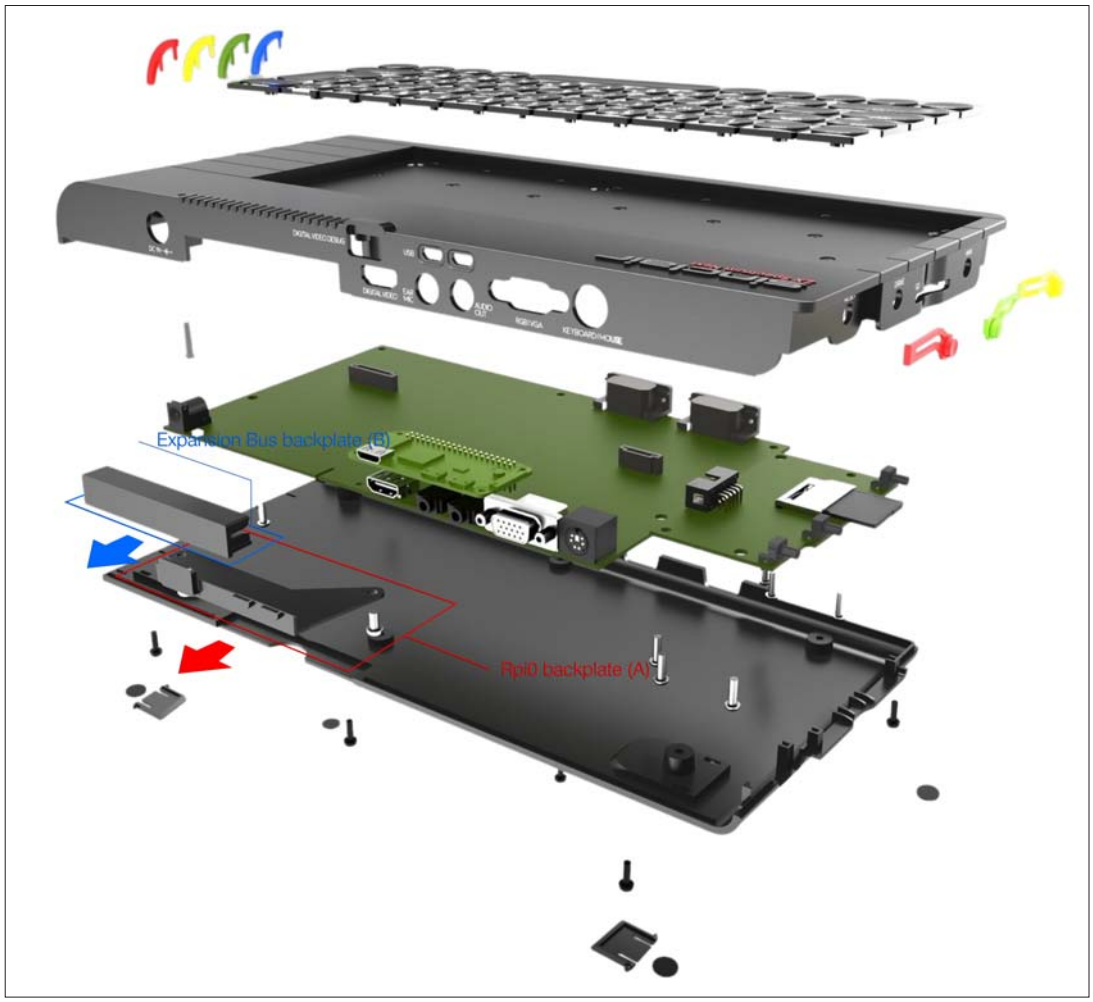



*Fig. 36 – Disassembled Next. Note the expansion port and Raspberry Pi 0 backplates*

The RPi0 installation on the Issue 4 mainboard does not differ in any way from the one done for the Issue 2. So if you don't have an Accelerated Next, follow the instructions in the previous section as they apply here as well.

**Notes**

The diagram above (*Fig. 36*) shows a completely disassembled ZX Spectrum Next computer for illustrative purposes only. Not everything can be disassembled; the keyboard itself is considered a standalone unit, fastened onto the top of the case by 10 M3 x 4mm screws and **YOU SHOULD NOT ATTEMPT TO DISASSEMBLE IT!!!**

**USER DISASSEMBLY of the Keyboard unit will NOT be supported by SpecNext Ltd and will invalidate the warranty!**

## Testing the add-ons' installation

Once you have your add-ons installed, it is time to test them; we'll start with the easier tests first and we'll progress to the most difficult ones.

## A. Testing the memory

This is by far the simplest test; if your memory installation worked, your *NextZXOS Startup menu* will report **1792K** instead of the **768K** it reported up until now (see *Fig. 37).*




*Fig. 37 – NextZXOS Options menu showing 2MB*

To further verify that the memory was properly installed, there's a program called **2MBTEST (v0.4c).nex** located under **c:/extras/memtest** in your **System/Next™** distribution.

Execute it with the browser or by using the **.nexload** dot command and let it go through all your memory testing it's working properly (see *Fig. 38 and 39*).




*Fig. 38 – Using the browser to launch 2MBTEST*




*Fig. 39 – 2MBTEST running without faults so far*

In case that something went wrong, your memory chips are either defective or you didn't install them properly. Make sure your memory chips are properly seated in their sockets by **a.** checking the space is left as in *Fig. 34* and **b.** pressing them firmly in their socket until you hear a subtle “click” sound. If the memory test still fails, your memory chips are probably defective.

## B. Testing the WiFi

Testing the WiFi feature is a very simple procedure. You only need to use *NextZXOS Startup Menu*, go under *Tools* and then select *WIFI setup*.

As seen in the figures below, you only have to select option **5** (Scan Networks) and once your network is found, you enter the password and that was all!

In case this does not work and you have still questions whether or if your WIFI module is really problematic, you can use the **.uart** dot command from your **System/Next™** distribution.




Fig. 40 – NextZXOS Tools submenu




Fig. 41 – WIFI setup program

**.uart** is not very complicated but it's quite temperamental especially if you use a PS/2 keyboard. You will need to use the standard ZX Spectrum keys; **CAPS SHIFT + 0** for **DELETE**, **SYMBOL SHIFT + K** for **+**, **SYMBOL SHIFT + C** for **?**, **SYMBOL SHIFT + P** for **"**, **SYMBOL SHIFT + N** for **,** and **SYMBOL SHIFT + L** for **=**.

You run it by issuing a:

```
.uart
```

you will be greeted by a screen full of information that will end in an **L** cursor. To test type the following:

```
AT
```

and press **ENTER**

If you're good so far, the *ESP* will be responding with:

```
OK
```

That's a very good sign. That means serial communications have been established. To see however if the *ESP* is actually working you'll need to issue a few more commands. Type:

```
AT+CWMODE?
```

the *ESP* there should respond with a **1**, **2** or **3** (this is the mode that's its working at; being **1** for Station, **2** for Access Point and **3** for both). Normally this should be enough to verify your *ESP* is working but if you want to take it one step further, you should set the *ESP* to station mode by giving:

```
AT+CWMODE=1
```

then check for what Access Points are around by doing:

```
AT+CWLAP
```

before finally connecting to one by giving the command:

```
AT+CWJAP="SSID","YourPass"
```

where *SSID* is the name of your network and *YourPass* is your WiFi password. The *ESP* will retain these so you can do if you want:

```
AT+RST
```

which will reset your *ESP* and give you a lot of information before concluding with a

```
WIFI CONNECTED
WIFI GOT IP
```

Exit .uart by pressing SYMBOL SHIFT + SPACE. If none of this worked, then the most likely culprits are that you either have a bad *ESP* module or that the power supply you're using is not powerful enough for both your ZX Spectrum Next and the *ESP* module. First try with a different power supply, otherwise return the ESP module for an exchange.

Note that different ESP firmware versions have slightly different versions of the commands above so always consult the most up-to-date documentation!

### C. Testing the RTC installation

There are several ways of testing if the *RTC* was properly installed the easiest of which is to launch again the *NextZXOS Startup Menu* and then go to the *Tools Submenu*. There apart from the *WIFI setup* option, you will find a *Set Clock* option. Launching it will allow you to program your RTC using just your cursor and the ENTER keys:




*Fig. 42 – NextZXOS Set Clock utility*

If everything worked right, then *NextZXOS* shall start reporting the current time and date on its menus like so:




*Fig. 43 – RTC Working*

If however the clock won't work, there's a very simple way of testing for the *RTC* and that's to give it the .time command. If it doesn't work outright, it will produce an output like the one shown in *Fig. 44*.

There are a few issues that can occur with the *RTC*; if the output is as above, the most likely culprit is the soldering of the IC onto the board. A cold solder will leave the *RTC* not working, a short somewhere will do the same but the *RTC* IC will start getting hot. If you feel the IC warming up disconnect all power immediately and inspect your soldering.

A defective battery holder installation as well as a defective (or depleted) battery will manifest itself with the *RTC* not keeping time upon bootup and *NextZXOS* not displaying the time and date information on its *Startup Menu*. Setting the time and date anew, however, will restore the time display.

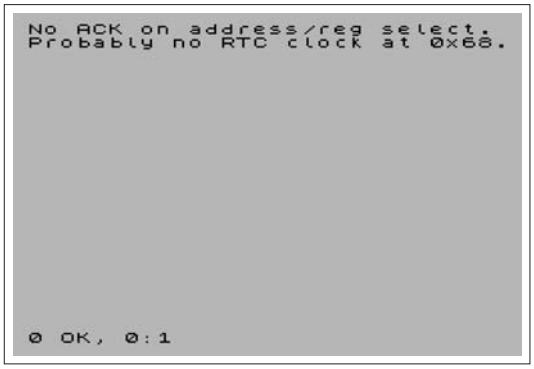



*Fig. 44 – RTC not working*

If on the other hand the commands described in the *Using the Real Time Clock hardware section* below do work, the time and date information appear in the *Startup menu* but time does not advance, then the issue lays usually with either the oscillator or the pins of the DS1307 IC these connect to. There's either a short somewhere or even a situation as simple as leftover flux from soldering. The oscillator can be damaged quite easily so make sure there's no continuity on its two legs before even turning the power on and inserting the battery in the holder.

## D. Testing the Accelerator installation

Before you can test that the accelerator is working, there are a few things you need to do: First find a **16** Gb or larger microSD Card and then you need to download the **NextPi2** distribution from: **http://zx.xalior.com/NextPi2** together with the instructions that accompany it.

Once you prepare the SD card according to the instructions, put it in your Pi Accelerator prior to booting up your ZX Spectrum Next. The Pi support programs are already in the **c:/apps/rpi** folder on your **System/Next™** distribution's SD, so you do not need to do anything else other than powering up the machine.

If you have access to the *RPi0* you should see the green led flashing while the ZX Spectrum Next is booting; that's a good first sign showing that the *RPi0* is loading its **NextPi2** distribution. The LED will eventually stop flashing and should turn into a steady green. Once you're all booted up, change to the **NextPi2** support folder, switch to the **terminex** folder and execute (with the browser or with the **SPECTRUM** command) **terminex.snx** by David Saphier. If the *RPi0* installation worked, you will see a message stating **Connection to NextPi established** followed by a **SUP>** prompt which means your *RPi0* installation was successful as shown in the figure below.




*Fig. 45 – RPi Supervisor prompt via Terminex*

If the SUP> prompt does not appear after a maximum of 20-25 seconds, that means there's something wrong. That doesn't mean your RPi0 is not working; especially if you saw the flashing green LED light on it earlier. This more than likely means that you didn't transfer the NextPI2 image properly or that there's some problem with the microSD card you used.

To verify the *RPi0* is working, you will need to unplug it from your ZX Spectrum Next, locate a micro usb power supply, an appropriate HDMI™ cable and a standard RPi0 distribution and power it independently.

If you can see output on the screen, then there's either a problem with your NextPI2 SD card (which you can verify by plugging its microSD card in the *RPi0*'s reader instead of the standard *RPi0* distribution), a cold solder on your IDC connector you soldered earlier, or finally, an insufficiently powerful, power supply for your ZX Spectrum Next.

The *RPi0s* are very resilient pieces of hardware and they don't fail easily; chances are any failure you experience is due to one of the cases listed.

## Using the Real Time Clock hardware

If you're lucky to have an expanded ZX Spectrum Next with the battery backed-up *Real Time Clock* (*RTC*) hardware installed (or if you followed the instructions to install it yourselves) then more options in timekeeping become available to you. These options do not suffer from the drawbacks and caveats laid out in the previous sections as this dedicated hardware option keeps time regardless of what else the computer is doing and in fact keeps time even when the computer is turned off.

The RTC is accessible via the function TIME$ (See *Chapter 17*) as well as two *dot commands*: .date and .time.

### Setting up your RTC for first use

As we saw above, invoking the Set Clock option found in the Tools submenu of the NextZXOS Startup Menu has the obvious benefit of setting up the clock AND testing at the same time. There are however two more ways to set your RTC up, especially if for some reason you wish to use an alternative to NextZXOS like for example esxDOS.

### Using .time and .date

This is very straightforward with the small exception that before you can use .date and .time you will need to set up your *Real Time Clock* hardware. Luckily this is only done once when you install it and whenever you need to change battery. You will initially (for safety) need to issue the command:

```
.time -di
```

This wipes the *RTC* signature from the chip and gets it ready to accept a date and time. You can then type:

```
.time "10:35:23"
```

where "10:35:23" can be substituted by any string of the format HH:MM:SS where HH (hour) is a number from 00 to 23 , MM (minute) is a number from 00 to 59 and SS (second) is a number from 00 to 59. You then enter the correct date by issuing:

```
.date "20/03/2023"
```

where "20/03/2023" can be substituted by any string of the format DD/MM/YYYY where DD is the day (01 to 31), MM is the month (01 to 12) and YYYY is any year from 2000 to 2099.

A few interesting things will happen once you install and setup your *RTC.* First, *NextZXOS* will report the time and date on its *Startup menu* (which is very nice indeed). Then, your saved files will start having a date and timestamp on them (visible with CAT EXP or .ls).

## Using the RTC together with the WiFi module

The RTC module is not very accurate and can lose several seconds over the period of a few weeks. Luckily, like other, much larger systems, the ZX Spectrum Next can also set its time from the internet, thanks to **.nxtp**, the dot command client to Robin Verhagen-Guest's *NeXt Time Protocol server*. Its syntax is:

**.nxtp** *server-address port [-z=Timezone]*

where *server-address* is a FQDN or IP address running a **nxtp** server, *port* is the port where that **nxtp** server is listening to (by default **12300**) and an optional *timezone* parameter to set the time to any location you would like from a list of acceptable timezones.

```
.nxtp time.zx.in.net 12300 -z=UTC
```

will talk to the the **nxtp** server located at **time.zx.in.net**, listening on port **12300** and set the RTC's time to **Coordinated Universal Time (UTC)** whereas

```
.nxtp time.zx.in.net 12300 -z=GMT
```

will do the same but for **Greenwich Mean Time** meaning the time will adjust for summer giving you **BST** and winter giving you **UTC**, as **.nxtp** already knows about *daylight savings*. It will work this into your *RTC*'s time setting meaning you never have to worry about setting your clock in the summer or winter provided your location observes these.

A full list of accepted timezones exists at the **.nxtp** project's wiki page located at: **https://github.com/Threetwosevensixseven/nxtp/wiki/Timezone-Codes**

It is a good idea, if you have an always working WiFi setup, to add **.nxtp** to your **autoexec.bas** file so it always sets the correct time whenever your ZX Spectrum Next boots. The potential startup delay is very small and the benefit of always having correct time outweighs the delay.

## Using the rest of the add-ons

Both the WiFi and Raspberry Pi Accelerator add-ons open up exciting features not before seen on a ZX Spectrum computer. This chapter provides only limited coverage as the feature set of both is still evolving. We have included all features implemented thus far (Audio playback, TZX loading, **DRIVER** support etc) in *Chapters 18, 19, 20* and this chapter, however, you're encouraged to read the accompanying documentation found in your **System/Next™** distribution and on **www.specnext.com** as they will always contain the most up-to-date information regarding these add-ons and newer ZX Spectrum Next features.

# Chapter 22 – IN, OUT and the Next Registers

We can instruct the processor to read from and write to memory by using **PEEK**, **POKE** and their variants. For all the possibilities, examine *Chapter 23 – The Memory*. The processor itself does not really care whether memory is ROM, RAM or even nothing at all; it just knows that there are **65536** memory addresses, and it can read a byte from each one, even if it's nonsense, and write a byte to each one, even if it gets lost because the address is read-only. In a completely analogous way, there are also **65536** *hardware* addresses, called I/O ports (Input/Output ports) that are separate from memory. These are used by the processor for communicating with attached devices like the keyboard or the display, and they can be controlled from *NextBASIC* by using the **IN** function and the **OUT** statement. There's a number of I/O ports, specific to the ZX Spectrum Next which control its advanced functions; they too, are accessible with **IN** and **OUT**, but two of them, are also accessible via a special dual statement/function, called **REG.**

## IN and OUT

**IN** is a function like the simplest form of **PEEK**:

**IN** *port*

It has one argument, the hardware address *port*, and its result is a byte read from that port. **OUT** on the other hand is a statement like a simple **POKE**:

**OUT** *port, v*

which writes value *v* to the hardware address *port*.

## Hardware address decoding

How the address is interpreted depends on the hardware in the computer and attached devices. In previous versions of the ZX Spectrum line of computers and especially in legacy peripherals, many different port addresses mapped to the same device. This is called *partial decoding* and happened because some address bits were ignored in the hardware to save on cost. As a consequence, entire ranges of port addresses were reserved by individual peripherals. This made it hard for new peripherals to find non-conflicting ports to use and, in reality, many did not and only managed to use ports that didn't conflict with the most popular peripherals. The situation was somewhat mitigated by the fact that only a couple of peripherals could be connected to the older ZX Spectrum machines at once, due to electrical limitations. Today, where modern ZX Spectrum implementations pack many devices into their hardware, this port conflict problem returns with renewed urgency, as any pair of devices with conflicting port addresses are not compatible with each other.

The ZX Spectrum Next *fully decodes* port addresses for new peripherals (meaning it does not ignore any address line), but because a lot of the hardware it contains is based on existing devices, those must continue to be partially decoded. In order to best understand the issues at hand, and in the table that follows which contains all available port addresses on the ZX Spectrum Next, it is best if we approach them as written in binary. That way we can easily show which bits are being ignored by a specific peripheral. Each hardware address is 16 bits wide, which we shall call (using **A** for address):

| A15 | A14 | A13 | A12 | A11 | A10 | A9 | A8 | A7 | A6 | A5 | A4 | A3 | A2 | A1 | A0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

Here **A0** is the 1st bit, **A1** the 2nd bit, **A2** the 3rd bit, **A3** the 4th bit and so on. The table that follows shows which bits are important for the corresponding device. For example, the ULA only needs A0 to be **0** in order to respond, which means it will respond to all **32768** even port addresses and not just its official port **254** (**FEh**). The byte read or written has **8** bits, and these are often referred to (using **D** for data) as:

| D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|

Here is a list of the port addresses used with their decoding. For the reason mentioned, only the ULA has an even port address and every even-numbered port **IN** will result in the ULA being read.

| R | W | A15 | A14 | A13 | A12 | A11 | A10 | A09 | A08 | A07 | A06 | A05 | A04 | A03 | A02 | A01 | A00 | Port (Hex) | Description |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ■ | ■ | | | | | | | | | | | | | | | | 0 | FEh | ULA |
| ■ | ■ | | | | | | | | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | FFh | Timex video, Floating bus |
| | ■ | 0 | | | | | | | | | | | | | | 0 | 1 | 7FFDh | Memory Paging Control |
| | ■ | 0 | 1 | | | | | | | | | | | | | 0 | 1 | 7FFDh | Memory Paging Control (+3/Next only) |
| | ■ | 1 | 1 | 0 | 1 | | | | | | | | | | | 0 | 1 | DFFDh | Next Memory Bank Select[1] |
| | ■ | 0 | 0 | 0 | 1 | | | | | | | | | | | 0 | 1 | 1FFDh | +3 Memory Paging Control |
| ■ | | 0 | 0 | 1 | 0 | | | | | | | | | | | 0 | 1 | 2FFDh | +3 FDC Status |
| ■ | ■ | 0 | 0 | 1 | 1 | | | | | | | | | | | 0 | 1 | 3FFDh | +3 FDC Control |
| | ■ | 1 | 1 | 1 | 0 | | | | | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | EFF7h | Pentagon 1024K Memory Paging Cntl. |
| ■ | | 0 | 0 | 0 | 0 | | | | | | | | | | | 0 | 1 | | +3 Floating bus |
| ■ | ■ | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 243Bh | NextREG Select |
| ■ | ■ | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 253Bh | NextREG Data |
| ■ | ■ | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 103Bh | I²C SCL |
| ■ | ■ | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 113Bh | I²C SDA |
| ■ | ■ | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 123Bh | Layer 2 |
| ■ | ■ | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 133Bh | UART Tx |
| ■ | ■ | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 143Bh | UART Rx |
| ■ | ■ | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 153Bh | UART Select |
| ■ | ■ | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 163Bh | UART Frame |
| ■ | ■ | 0 | 0 | 0 | 1 | 1 | | | | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 183Bh | CTC Channel 0[2] |
| ■ | ■ | 0 | 0 | 0 | 1 | 1 | | | | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 193Bh | CTC Channel 1[2] |
| ■ | ■ | 0 | 0 | 0 | 1 | 1 | | | | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1A3Bh | CTC Channel 2[2] |
| ■ | ■ | 0 | 0 | 0 | 1 | 1 | | | | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1B3Bh | CTC Channel 3[2] |
| ■ | ■ | 0 | 0 | 0 | 1 | 1 | | | | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1C3Bh | CTC Channel 4[2] |
| ■ | ■ | 0 | 0 | 0 | 1 | 1 | | | | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1D3Bh | CTC Channel 5[2] |
| ■ | ■ | 0 | 0 | 0 | 1 | 1 | | | | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1E3Bh | CTC Channel 6[2] |
| ■ | ■ | 0 | 0 | 0 | 1 | 1 | | | | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1F3Bh | CTC Channel 7[2] |
| | ■ | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | BF3Bh | ULAplus Register |
| ■ | ■ | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | FF3Bh | ULAplus Data |
| ■ | ■ | | | | | | | | | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0Bh | z80DMA[3] |
| ■ | ■ | | | | | | | | | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 6Bh | zxnDMA[3] |
| ■ | ■ | 1 | 1 | | | | | | | | | | | | 1 | 0 | 1 | FFFDh | AY Register |
| ■ | ■ | 1 | 0 | | | | | | | | | | | | 1 | 0 | 1 | BFFDh | AY Data (readable on +3/Next only) |
| ■ | | 1 | 0 | | | | | | | | | | | 0 | 1 | 0 | 1 | BFF5h | AY Inf (inside BFFDh decoding) |
| | ■ | | | | | | | | | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1Fh | DAC A |
| | ■ | | | | | | | | | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | F1h | DAC A[4] |
| | ■ | | | | | | | | | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 3Fh | DAC A |
| | ■ | | | | | | | | | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0Fh | DAC B |
| | ■ | | | | | | | | | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | F3h | DAC B |
| | ■ | | | | | | | | | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | DFh | DAC A,D |
| | ■ | | | | | | | | | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | FBh | DAC A,D |
| | ■ | | | | | | | | | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | B3h | DAC B,C |
| | ■ | | | | | | | | | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 4Fh | DAC C |
| | ■ | | | | | | | | | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | F9h | DAC C[4] |
| | ■ | | | | | | | | | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 5Fh | DAC D |
| | ■ | | | | | | | | | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | E7h | SPI CS (SD card, Flash, RPi0) |
| ■ | ■ | | | | | | | | | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | EBh | SPI DATA |
| ■ | ■ | | | | | | | | | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | E3h | divMMC control |

1 *Precedence over AY*
2 *Temporarily reduced to first four CTC channels only*
3 *Selecting either DMA port, also selects the mode in which the DMA operates*
4 *Precedence over xxFD*

| R | W | A15 | A14 | A13 | A12 | A11 | A10 | A09 | A08 | A07 | A06 | A05 | A04 | A03 | A02 | A01 | A00 | Port (Hex) | Description |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ■ | | | | | | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | FBDFh | KEMPSTON Mouse X |
| ■ | | | | | | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | FFDFh | KEMPSTON Mouse Y |
| ■ | | | | | | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | FADFh | KEMPSTON Mouse Wheel, Buttons |
| ■ | | | | | | | | | | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1Fh | KEMPSTON Joystick 1 |
| ■ | | | | | | | | | | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | DFh | KEMPSTON Joystick 1 Alias |
| ■ | | | | | | | | | | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 37h | KEMPSTON Joystick 2 |
| ■ | ■ | | | | | | | | | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1Fh | Multiface 1 Disable |
| ■ | ■ | | | | | | | | | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 9Fh | Multiface 1 Enable |
| ■ | ■ | | | | | | | | | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1Fh | Multiface 128 v87.12 Disable |
| ■ | ■ | | | | | | | | | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 9Fh | Multiface 128 v87.12 Enable |
| ■ | ■ | | | | | | | | | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 3Fh | Multiface 128 v87.2 Disable |
| ■ | ■ | | | | | | | | | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | BFh | Multiface 128 v87.2 Enable |
| ■ | ■ | | | | | | | | | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | BFh | Multiface +3 Disable |
| ■ | ■ | | | | | | | | | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 3Fh | Multiface +3 Enable |
| ■ | ■ | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 303Bh | Sprite slot flags |
| | ■ | | | | | | | | | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 57h | Sprite Attributes |
| | ■ | | | | | | | | | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 5Bh | Sprite Pattern |

The above table describes the ports that are used to control and communicate with additional hardware features of the ZX Spectrum Next. Therein exist 17 ports that are of special signifance as they are unique to the ZX Spectrum Next. These are:

| Port Name | Address (Hex) | Address (Dec) | Description |
|---|---|---|---|
| NextReg Select | 243Bh | 9275 | Communicates with the ZX Spectrum Next hardware |
| NextReg Data | 253Bh | 9531 | Used after register selection to send and read data |
| I²C SCL | 103Bh | 4155 | Used for I²C device communication (RTC etc) |
| I²C SDA | 113Bh | 4411 | Sends/Receives data from/to the I²C bus |
| Layer 2 | 123Bh | 4667 | Used to control Layer 2 |
| UART Tx | 133Bh | 4923 | Transmits data from the UART |
| UART Rx | 143Bh | 5179 | Receives data from the UART |
| UART Select | 153Bh | 5435 | Selects which UART is in use |
| UART Frame | 163Bh | 5691 | Sets up the UART framing |
| CTC | 183Bh - 1FB3h | 6203 - 8115 | Configures the Counter Timer Circuits |
| z80DMA | 0Bh | 11 | Programs the DMA in a z80DMA compatible mode |
| zxnDMA | 6Bh | 107 | Programs the DMA in zxnDMA mode |
| SPI Select | E7h | 231 | Selects an SPI peripheral (SD Card/Flash ROM/RPi0) |
| SPI DATA | EBh | 235 | Sends/Receives Data via the SPI bus a byte at a time |
| SPRITE | 303Bh | 12347 | Controls the Next Sprite Engine |
| SPRITE ATTR | 57h | 87 | Sends Sprite Attributes to the Sprite Engine |
| SPRITE PATTERN | 5Bh | 91 | Sends Sprite Pattern to the Sprite Engine |

Two of the most important ports in this list are collectively called *Next Register* or *NextREG* for short. Most of the machine's features can be controlled through *NextREG*.

## Accessing the ZX Spectrum Next features with NextREG

We use a *NextREG* by first selecting it with the control port and then writing to – or reading from – the *NextREG* data port.

In *NextBASIC* this is achieved with two consecutive **OUT** commands in the case of writing or with a combination of consecutive **OUT** and **IN** in the case of reading from a NextREG.

The first command is directed to the *Select* port **9275** (**243Bh**), selecting a specific register and the second to the *Data* port **9531** (**253Bh**) to modify or read the value stored there.

If given from *NextBASIC*, these commands must be given consecutively in one line as *NextBASIC* may do something different with *NextREG* in-between commands. If you give the first and then wait to give the second, *NextBASIC* may have changed the *Select register* in the meantime; so by giving them together you give it no time to do something else.

The Z80N CPU which powers the ZX Spectrum Next, also provides a special **NEXTREG** instruction and this is referenced in *Appendix A*.

Finally, as mentioned in the introduction, in order to read *NextREG, NextBASIC* also has a specialised command and function called **REG**, which is much easier to use than the combination of **OUT** and **IN** keywords.

We'll showcase both methods here, in order for you to be able to use either as there are cases where the ZX Spectrum Next's facilities are still available but without *NextBASIC* to provide access to them. The command as a statement has the form:

**REG** *n,v*

which is essentially the same as doing:

**OUT 9275**, *n*:**OUT 9531**, *v*

Obviously *n* is the register number and *v* is the value we modify the register with. As a function, **REG** has the following form:

**% REG** *n* or **REG** *n*

as it works with both the integer as well as the standard expression evaluators. This essentially is the same as executing

**OUT 9275, n: %x = % IN 9531** (or the equivalent **x = IN 9531**).

Let's give one simple example in both forms and let's mix-and-match a bit as well, to show the equivalency:

Assuming we want to change speeds to **28MHz**, we could give:

```
RUN AT 3
```

or

```
OUT 9275, 7:OUT 9531, 3
```

and we can verify that it is set by either bringing up any *NextBASIC* menu (menus list the currently set speed on top) with the **EDIT** key or by doing:

```
OUT 9275,7: PRINT % IN 9531 & @11
```

Which is the same as

```
PRINT % REG 7&@11
```

You can verify this actually changes things by doing a **RUN AT 2** and give the **OUT**/**IN** sequence again. As you will see from the list that follows, not every *NextREG* is dedicated solely to one function; in this case the only bits that concerned us were Bits **0** and **1** and that's why we used a 2-bit bitmask with the bitwise *AND* operator **&**. For the same example using just the **REG** command, our line would have been as simple as:

```
REG 7,3
```

In our example in *Chapter 17* where we read *NextREG* 5h we also used bit shifting which is a great way to get the value of a single bit in a register. In our case we only needed bit 2 of the register so after getting the specific bit by bitwise AND (**&**) the register value with a 3-bit bitmask, we shifted it two places (bits) to the right by using the right bit-shifting operator (**>>**). That way we were able to get the value of the single register bit.

Generally speaking, you will often want to modify individual bits in a *NextREG* without changing the remaining ones.

You can do this by first reading the NextREG and then *masking off* the bits you want to leave unchanged by using bitwise AND (**&**) and finally write that value with the new bits added in.

We'll list all of *NextREG*s below in numerical order. Not every register is accessible, so pay attention to the key at the start of the list to understand whether a register can be read, written or both.

## Next Register Diagrams' Key

| | |
|---|---|
| | Reserved value in either R(ead) or W(rite) condition but used in the inverse – **Not Applicable** |
| | Reserved value - **Not Applicable** |
| X | Reserved value **MUST be X** |
| | **Not Applicable / Don't care** |
| R | **Read** (if marked). Unmarked means **Not Applicable** |
| W | **Write** (if marked). Unmarked means Not Applicable |
| H | **Hard** Reset / **Soft** Reset / **Config Mode**. Unmarked means **Soft Reset** if there's a value in column **D** |
| D | Contains the default value after a reset (Soft, Hard or Config as marked in column H).* refers to notes below |
| ■ | In columns R/W marks the status of the register. In column H means Hard Reset |
| ◆ | Means **Config Mode** |
| ● | Means *any* value (**0** or **1**) |
| N | Any letter in a data bit position refers you to the notes below |

## NextREG 00 (00h) – Machine ID

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Machine ID | ■ | | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | EMULATORS | | |
| | | | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | ZX Spectrum Next | | |
| | | | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | ZX Spectrum Next Anti-Brick | | |
| | | | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | Next Core on UnAmiga Reloaded | | |
| | | | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | Next Core on UnAmiga | | |
| | | | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | Next Core on SiDi | | |
| | | | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | Next Core on MiST | | |
| | | | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | Next Core on MiSTer | | |
| | | | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | Next Core on ZX-DOS/gomaDOS | | |

## NextREG 01 (01h) – Core Version

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Minor Version Number | ■ | | | | | | ● | ● | ● | ● | Minor Version | | |
| Major Version Number | ■ | | | ● | ● | ● | ● | | | | Major Version | | |

See **NextREG 14** (**0Eh**) for sub minor version number

## NextREG 02 (02h) – Reset

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Last System Reset Type | ■ | | | | | | | | 0 | 1 | Soft Reset | | |
| | | | | | | | | | 1 | 0 | Hard Reset | | |
| divMMC NMI source | ■ | | | | | | | 0 | | | divMMC NMI not generated by NR 02h | | |
| | | | | | | | | 1 | | | divMMC NMI generated by NR 02h | | |
| MF NMI source | ■ | | | | | | 0 | | | | MF NMI not generated by NR 02h | | |
| | | | | | | | 1 | | | | MF NMI generated by NR 02h | | |
| | | | | | | 0 | | | | | MF NMI not generated by I/O Trap[2].* | | |
| | | | | | | 1 | | | | | MF NMI generated by I/O Trap[2].* | | |
| Reserved | ■ | | | | | | | | | | Reserved | | |
| ESP/Expansion Bus RESET flag | ■ | | 0 | | | | | | | | RESET not asserted | ■ | 0 |
| | | | 1 | | | | | | | | RESET asserted | | |
| Generate System Reset | ■ | | | | | | | | 0 | 1 | Generate Soft Reset | | |
| | | | | | | | | | 1 | 0 | Generate Hard Reset (reboot) | | |
| divMMC NMI control | ■ | | | | | | | ● | | | Generate/Clear divMMC NMI[3] | | |
| MF NMI control | ■ | | | | | | ● | | | | Generate/Clear MF NMI[3] | | |
| | | | | | | ● | | | | | If 0 Clear MF I/O Trap[2,3] | | |
| Reserved | ■ | | 0 | 0 | | | | | | | Reserved (must be 0) | | |
| Generate ESP/Exp. Bus Reset[1] | ■ | ● | | | | | | | | | Generate/Release Reset (Exp. Bus & ESP) | | |

1 A full reset cycle for the ESP and Expansion Bus, requires setting **D7** first to **1** for at **least 100 ms** and then to **0**. The 100 ms assertion is required by the ESP module. If not explicitly released the Expansion Bus and ESP will stay with RESET asserted until the next system hard reset
Of **D0 – D1** Hard Reset has precedence
2 Experimental
3 These signals are ignored if the Multiface, divMMC, DMA or external NMI master is active, Copper cannot clear these bits. Note: An I/O trap could occur at the same time as MF / divMMC cause; always check this bit in NMI ISR if important
* See **NextREG 218** (**DAh**)

## NextREG 03 (03h) – Machine Type

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Machine Type | ■ | ■ | | | | | | 0 | 0 | 0 | Configuration mode | | |
| | | | | | | | | 0 | 0 | 1 | ZX 48K | | |
| | | | | | | | | 0 | 1 | 0 | ZX 128K / +2 | ◆ | |
| | | | | | | | | 0 | 1 | 1 | ZX +2A / +2B / +3 / Next | | |
| | | | | | | | | 1 | 0 | 0 | Pentagon Clones* | | |
| Display Timing user lock control | ■ | | | | | | 0 | | | | No User Lock on display timing applied | | |
| | | | | | | | 1 | | | | User lock on display timing applied | | |
| | ■ | | | | | | ● | | | | Toggle User Lock on Display Timing | ■ | 0 |
| Display Timing | ■ | ■ | | 0 | 0 | 0 | | | | | Internal Use | | |
| | | | | 0 | 0 | 1 | | | | | ZX 48K | | |
| | | | | 0 | 1 | 0 | | | | | ZX 128K / +2 | | |
| | | | | 0 | 1 | 1 | | | | | ZX +2A / +2B / +3 / Next | | |
| | | | | 1 | 0 | 0 | | | | | Pentagon* | | |
| NR 68 (44h) 2nd Byte Indicator | ■ | ● | | | | | | | | | Palette Entry (NR 68) Second Byte flag | | 0 |
| Display Timing change enable | ■ | ● | | | | | | | | | Allow changes to D4:6 | | 0 |

* Pentagon timing is **50 Hz** only
Machine type determines which ROMs are loaded and Display Timing also affects port decoding

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 16K RAM bank mapping | ■ | | ● | ● | ● | ● | ● | ● | ● | ● | Maps a 16K RAM Bank no. (0–127)* | ◆ | 0 |
| Reserved | ■ | | 0 | | | | | | | | Reserved, must be 0 | | |

* Maps a 16K RAM bank over the **bottom** 16K. Applies only in configuration mode when the boot rom is disabled
Even multiples of 256K are unreliable if storing data in RAM for the Next Core started

## NextREG 05 (05h) – Peripheral 1 Settings

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Scandoubler[1] | ■ | ■ | | | | | | | | 0 | Scandoubler Disabled | | |
| | | | | | | | | | | 1 | Scandoubler Enabled | | |
| Vertical Frequency | ■ | ■ | | | | | | | 0 | | 50 Hz mode | | |
| | | | | | | | | | 1 | | 60 Hz mode* | | |
| Joystick 1** | ■ | ■ | 0 | 0 | | | 0 | | | | Sinclair 2 (12345) | | |
| | | | 0 | 0 | | | 1 | | | | Kempston 2 (Port 37h) | | |
| | | | 0 | 1 | | | 0 | | | | Kempston 1 (Port 1Fh) | | |
| | | | 0 | 1 | | | 1 | | | | MD 1 (Port 1Fh) | | |
| | | | 1 | 0 | | | 0 | | | | Cursor (56780) | | |
| | | | 1 | 0 | | | 1 | | | | MD 2 (Port 37h) | | |
| | | | 1 | 1 | | | 0 | | | | Sinclair 1 (67890) | | |
| | | | 1 | 1 | | | 1 | | | | User Defined Keys Joystick | | |
| Joystick 2** | ■ | ■ | | | 0 | 0 | | | | 0 | Sinclair 2 (12345) | | |
| | | | | | 0 | 0 | | | | 1 | Kempston 2 (Port 37h) | | |
| | | | | | 0 | 1 | | | | 0 | Kempston 1 (Port 1Fh) | | |
| | | | | | 0 | 1 | | | | 1 | MD 1 (Port 1Fh) | | |
| | | | | | 1 | 0 | | | | 0 | Cursor (56780) | | |
| | | | | | 1 | 0 | | | | 1 | MD 2 (Port 37h) | | |
| | | | | | 1 | 1 | | | | 0 | Sinclair 1 (67890) | | |
| | | | | | 1 | 1 | | | | 1 | User Defined Keys Joystick | | |

1 Enabled for VGA / HDMI, Disabled for RGB mode
* Pentagon Clones have no 60 Hz mode; when in Pentagon mode, every setting is **50 Hz**.
** Joystick ports can be placed into I/O mode via **NextREG 11** (**Bh**)
Programming the user defined keys joystick is done through the PS/2 keymap interface on **NextREG 40** (**28h**), **41** (**29h**) and **43** (**2Bh**):
1. Write **128** to **NextREG 40** (**28h**)
2. Write **0** (Left joystick) or **16** (Right joystick) to **NextREG 41** (**29h**)
3. Write **12** bytes to **NextREG 43** (**2Bh**) in order. The bytes correspond to the twelve buttons on an MD pad as follows (**Byte 0** to **Byte 11**):
**R, L, D, U, B, C, A, START, Y, Z, X, MODE**
4. Each byte written identifies a key in the **8x7** membrane; **D5:3** select the row and **D2:0** the column with **111** meaning no action
In Kempston and MD modes, excess buttons on an controller not read via ports will generate key input if so programmed

## NextREG 06 (06h) – Peripheral 2 Settings

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| PSG Mode Control | ■ | ■ | | | | | | | 0 | 0 | YM | | |
| | | | | | | | | | 0 | 1 | AY | | |
| | | | | | | | | | 1 | 1 | Hold all PSGs in Reset | | |
| PS/2 Mode Control | ■ | ■ | | | | | | | 0 | | Keyboard Primary | | |
| | | | | | | | | | 1 | | Mouse Primary | ◆ | 0 |
| NMI Button Control | ■ | ■ | | | | | | ● | | | NMI button enable* | ■ | 0 |
| divMMC NMI Control | ■ | ■ | | | | | ● | | | | divMMC NMI** button enable | ■ | 0 |
| F3 Hotkey Control | ■ | ■ | | | | ● | | | | | 50 Hz / 60 Hz hotkey enable | | 1 |
| Internal Speaker BEEP Control | ■ | ■ | | | 0 | | | | | | All audio copied to speaker | ■ | 0 |
| | | | | | 1 | | | | | | Only BEEP sounds diverted to speaker | | |
| F5, F6 and F8 Hotkey Control | ■ | ■ | ● | | | | | | | | CPU Speed and Exp. Bus hotkeys enable | | 1 |

* NMI button refers to the button to the right side of the SD Card reader
** Refers to the Drive button to the left side of the SD Card reader

## NextREG 07 (07h) – CPU Speed

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| CPU Speed Control[1] | ■ | ■ | | | | | | | 0 | 0 | 3.5 MHz | | * |
| | | | | | | | | | 0 | 1 | 7 MHz | | |
| | | | | | | | | | 1 | 0 | 14 MHz | | |
| | | | | | | | | | 1 | 1 | 28 MHz | | |
| Reserved | ■ | | 0 | 0 | 0 | 0 | 0 | 0 | | | Reserved, must be 0 | | |
| Reserved | ■ | | 0 | 0 | | | 0 | 0 | | | Reserved, must be 0 | | |
| Current Actual[2] CPU Speed | ■ | | | | 0 | 0 | | | | | 3.5 MHz | | |
| | | | | | 0 | 1 | | | | | 7 MHz | | |
| | | | | | 1 | 0 | | | | | 14 MHz | | |
| | | | | | 1 | 1 | | | | | 28 MHz | | |

* Soft reset defaults this to **00**
1 When read returns the Programmed CPU speed as it may differ from actual speed. See below
2 Current Actual speed may differ from the set speed due to Expansion Bus use, or another forced change

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Issue 2 keyboard | ■ | ■ | | | | | | | | ● | Enable Issue 2 keyboard | ■ | 0 |
| NextSound | | | | | | | | | ● | | Enable Multiple PSGs* | ■ | 0 |
| Timex Video Port Control | ■ | ■ | | | | | | ● | | | Enable read of Port FFh (Timex)** | ■ | 0 |
| DACs Control | | | | | | | ● | | | | Enable DACs (A-B-C-D) | ■ | 0 |
| Internal Speaker Control | ■ | ■ | | | | | ● | | | | Enable Internal Speaker | ■ | 1 |
| PSG Stereo Mode Control | ■ | ■ | | | | 0 | | | | | Select ABC | ■ | 0 |
| | | | | | | 1 | | | | | Select ACB | | |
| Contention Control | ■ | ■ | | | ● | | | | | | Disable RAM and Port Contention | | 0 |
| 128K Banking Unlock Control | ■ | ■ | ● | | | | | | | | Unlock Port 7FFDh **D5** (Unlocked = 1) | | 0 |

* Currently selected PSG is frozen when disabled
** Enabling Timex Video Port read hides the floating bus on **FFh**

## NextREG 09 (09h) – Peripheral 4 Settings

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Scanline Strength | ■ | ■ | | | | | | | 0 | 0 | Scanlines off | | |
| | | | | | | | | | 0 | 1 | Scanlines at 50% | | |
| | | | | | | | | | 1 | 0 | Scanlines at 25% | | |
| | | | | | | | | | 1 | 1 | Scanlines at 12.5% | | |
| HDMI audio output Control | ■ | ■ | | | | | | ● | | | HDMI audio mute | ■ | 0 |
| divMMC mapRAM bit Control | ■ | ■ | | | | | ● | | | | Reset bit 6 port E3h (Read is always 0) | | |
| Sprite Lockstep Control | ■ | ■ | | | | | ● | | | | Enable Sprite ID Lockstep | | 0 |
| PSG 0 Mono Mode Control | ■ | ■ | | | | ● | | | | | Enable Mono | ■ | 0 |
| PSG 1 Mono Mode Control | ■ | ■ | | | ● | | | | | | Enable Mono | ■ | 0 |
| PSG 2 Mono Mode Control | ■ | ■ | ● | | | | | | | | Enable Mono | ■ | 0 |

In Sprite Lockstep, **NextREG 52 (34h)** and Port **12347 (303Bh)** are in Lockstep

## NextREG 10 (0Ah) – Peripheral 5 Settings

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Mouse resolution Control | ■ | ■ | | | | | | | 0 | 0 | Low DPI | | |
| | | | | | | | | | 0 | 1 | Default | ■ | |
| | | | | | | | | | 1 | 0 | Medium DPI | | |
| | | | | | | | | | 1 | 1 | High DPI | | |
| Reserved | ■ | ■ | | | | | | 0 | | | Reserved, must be 0 | | 0 |
| Mouse button swap Control | ■ | ■ | | | | | ● | | | | Reverses (swaps) L & R mouse buttons | ■ | 0 |
| divMMC Automap Control | ■ | ■ | | | | ● | | | | | Enables divMMC Automap | ■ | 0 |
| Reserved | ■ | ■ | | | 0 | | | | | | Reserved, must be 0 | ■ | 0 |
| Multiface Type | ■ | ■ | 0 | 0 | | | | | | | Multiface +3[1] | ◆ | 0 |
| | | | 0 | 1 | | | | | | | Multiface 128 v87.2[2] | | |
| | | | 1 | 0 | | | | | | | Multiface 128 v87.12[3] | | |
| | | | 1 | 1 | | | | | | | Multiface 1[4] | | |

[1] Enables port 3Fh, disables port BFh
[2] Enables port BFh, disables port 3Fh
[3] Enables port 9Fh, disables port 1Fh
[4] Enables port 9Fh, disables port 1Fh

## NextREG 11 (0Bh) – Joystick Port Mode Selection

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Parameter (Bit Bang) | ■ | ■ | | | | | | | | ● | Copied to pin 7 of the active connector[1] | | |
| Parameter (Clock) | | | | | | | | | | 0 | Hold high when clock becomes high | | |
| | | | | | | | | | | 1 | Run[2] | | 1 |
| Parameter (UART) | | | | | | | | | | 0 | Redirect ESP UART_0 to connector[3] | | |
| | | | | | | | | | | 1 | Redirect Pi UART_1 to connector[3] | | |
| Reserved | | | | | | | | 0 | 0 | 0 | Reserved, must be 0 | | 0 |
| I/OMode Control | | | | | | 0 | 0 | | | | Bit Bang | | |
| | | | | | | 0 | 1 | | | | Clock | | 0 |
| | | | | | | 1 | 0 | | | | UART on left joystick port | | |
| | | | | | | 1 | 1 | | | | UART on right joystick port | | |
| Reserved | | | | | 0 | | | | | | Reserved, must be 0 | | 0 |
| Joystick Port Mode Selection | | | 0 | | | | | | | | Joysticks Enabled | | 0 |
| | | | 1 | | | | | | | | I/O Mode Enabled | | |

[1] The state of output pin 7 is stored internally in a register and is retained across changing modes and while I/O is disabled
[2] CTC Ch. 3 is currently used to drive pin 7 in clock mode. Frequency= $f_{CPU} \div 2$
[3] Tx out on pin 7, Rx in from pin 9, CTS_n in from pin 6. CTS_n is only active if the selected UART is in hardware flow control mode
* While in I/O mode, keyboard joystick types (Sinclair, Cursor etc) produce no readings but the current state of pins can still be read via the Kempston ports. When leaving I/O mode, joystick operation resumes after approximately 64 scanlines have passed

## NextREG 14 (0Eh) – Core Version (Sub minor number)

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Sub Minor Number | ■ | | ● | ● | ● | ● | ● | ● | ● | ● | Core Sub Minor Version Number | | |

See **NextREG 01 (01h)** for Major and Minor Core Version

## NextREG 15 (0Fh) – Board ID

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Board Revision | ■ | | | | | | 0 | 0 | 0 | 0 | ZX Spectrum Next Issue 2[1] | | |
| | | | | | | | 0 | 0 | 0 | 1 | ZX Spectrum Next Issue 3[1] | | |
| | | | | | | | 0 | 0 | 1 | 0 | ZX Spectrum Next Issue 4[2] | | |
| Reserved | ■ | | 0 | 0 | 0 | 0 | | | | | Reserved | | |

[1] XC6SLX16-2FTG256, 128 Mbit W25Q128JV Flash, 24-bit SPI, 64K*8 Core size
[2] XC7A15T-1CSG324, 256 Mbit MX25L25645G Flash, 32-bit SPI, 64K*34 Core size

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| NMI Button State Flag | ■ | | | | | | | | | ● | NMI Button Pressed | |
| Drive Button State Flag | ■ | | | | | | | | | ● | Drive Button Pressed | |
| | ■ | | ● | ● | ● | ● | ● | | | | Core ID (0-31) | |
| Reserved | ■ | | 0 | | | | | | | | Reserved, must be 0 | |
| Core ID | ■ | | | ● | ● | ● | ● | ● | ● | ● | Core ID (0-31) | ◆ |
| Reserved | ■ | | | 0 | 0 | | | | | | Reserved, must be 0 | |
| Start Core | ■ | ● | | | | | | | | | Reboot FPGA using selected core | |

Core ID with **D0** through **D4** can be set in configuration mode only

## NextREG 17 (11h) – Video Timing

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| VGA Timing | ■ | ■ | | | | | | | 0 | 0 | 0 | Base VGA timing, clk28 = 28000000 | ◆ |
| | | | | | | | | | 0 | 0 | 1 | VGA setting 1, clk28 = 28571429 | |
| | | | | | | | | | 0 | 1 | 0 | VGA setting 2, clk28 = 29464286 | |
| | | | | | | | | | 0 | 1 | 1 | VGA setting 3, clk28 = 30000000 | |
| | | | | | | | | | 1 | 0 | 0 | VGA setting 4, clk28 = 31000000 | |
| | | | | | | | | | 1 | 0 | 1 | VGA setting 5, clk28 = 32000000 | |
| | | | | | | | | | 1 | 1 | 0 | VGA setting 6, clk28 = 33000000 | |
| | | | | | | | | | 1 | 1 | 1 | Digital, clk28 =27000000 | |
| Reserved | ■ | ■ | 0 | 0 | 0 | 0 | 0 | 0 | | | | Reserved, must be 0 | |

50 Hz / 60 Hz depends on **NextREG 05 (05h):D2**
NextREG writable in configuration mode only

## NextREG 18 (12h) – Layer 2 Active RAM Bank

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Layer 2 Active RAM Bank | ■ | ■ | ● | ● | ● | ● | ● | ● | ● | ● | Starting 16K RAM Bank | |
| Reserved | ■ | ■ | 0 | | | | | | | | Reserved, must be 0 | |

Soft reset resets the default to **8**. NextZXOS changes that to **9**

## NextREG 19 (13h) – Layer 2 Shadow RAM Bank

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Layer 2 Shadow RAM Bank | ■ | ■ | ● | ● | ● | ● | ● | ● | ● | ● | Starting 16K RAM Bank | |
| Reserved | ■ | ■ | 0 | | | | | | | | Reserved, must be 0 | |

Soft reset resets the default to **11**. NextZXOS changes that to **12**

## NextREG 20 (14h) – Global Transparency Colour

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Global Transparency Mask | ■ | ■ | ● | ● | ● | ● | ● | ● | ● | ● | 8-bit colour value | |

Default value upon soft reset is **E3h (227)**
This value is 8-bit; the transparency colour is compared against the MSB of the actual 9-bi
colour; as such, two colours (with either value of $B_0$) are made transparent
This setting only applies to Layer 2, Layer 0 and Layer 1. Sprites use **NextREG 75 (4Bh)** and Layer 3 uses **NextREG 76 (4Ch)** for transparency except in text mode

## NextREG 21 (15h) – Sprite and Layer System Setup

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Sprite Engine Control | ■ | ■ | | | | | | | | ● | Enable Sprites | |
| Sprite Extended Area Control | ■ | ■ | | | | | | | | ● | Enable Sprites over border | |
| Set Layer Priority | ■ | ■ | | | | | | 0 | 0 | 0 | S L U | |
| | | | | | | | | 0 | 0 | 1 | L S U | |
| | | | | | | | | 0 | 1 | 0 | S U L | |
| | | | | | | | | 0 | 1 | 1 | L U S | |
| | | | | | | | | 1 | 0 | 0 | U S L | |
| | | | | | | | | 1 | 0 | 1 | U L S | |
| | | | | | | | | 1 | 1 | 0 | (U\|T)S(T\|U)(B+L) combined[1] | |
| | | | | | | | | 1 | 1 | 1 | (U\|T)S(T\|U)(B+L) combined[1] | |
| Sprite Border Clipping | ■ | ■ | | | | | ● | | | | Enable Sprite Clipping in over border mode | |
| Sprite Priority | ■ | ■ | | | | 0 | | | | | Sprite 127 on top | |
| | | | | | | 1 | | | | | Sprite 0 on top | |
| LoRes Control | ■ | ■ | ● | | | | | | | | Enable LoRes | |

[1] Colours Clamped to [0,7]
ULA means all ULA modes: Layers 0 and all Layer 1 types. Value upon soft reset is **000**

## NextREG 22 (16h) – Layer 2 Horizontal Scroll Control LSB

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| X Offset LSB | ■ | ■ | ● | ● | ● | ● | ● | ● | ● | ● | 8-bit value of X Offset LSB (0 – 511) | |

MSB is in **NextREG 113 (71h)**

## NextREG 23 (17h) – Layer 2 Vertical Scroll Control

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Y Offset | ■ | ■ | ● | ● | ● | ● | ● | ● | ● | ● | 8-bit value of Y Offset | |

## NextREG 24 (18h) – Layer 2 Clip Window Definition

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Coordinate ($X_{beg}$,$X_{end}$,$Y_{beg}$,$Y_{end}$) | ■ | ■ | ● | ● | ● | ● | ● | ● | ● | ● | 8-bit value of $X_{beg}$,$X_{end}$,$Y_{beg}$,$Y_{end}$ coordinate[1] | |

[1] 1st Write $X_{beg}$ position – *Resets to **0**
2nd Write $X_{end}$ position – *Resets to **255**
3rd Write $Y_{beg}$ position – *Resets to **0**
4th Write $Y_{end}$ position – *Resets to **191**
Reads do not advance the clip coordinate – Use **NR 28 (1Ch)** to find out the current clip coordinate or to set the clip coordinate to $X_{beg}$. The **X** coordinates are internally doubled fo higher resolution modes. This means that $X_{end}$ =159 covers the full **320** resolution.

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Coordinate ($X_{beg}$,$X_{end}$,$Y_{beg}$,$Y_{end}$) | ■ | ■ | ● | ● | ● | ● | ● | ● | ● | ● | 8-bit value $X_{beg}$-$X_{end}$,$Y_{beg}$-$Y_{end}$ coordinate[1] | | * |

1 1st Write $X_{beg}$ position – *Resets to **0**
2nd Write $X_{end}$ position – *Resets to **255**
3rd Write $Y_{beg}$ position – *Resets to **0**
4th Write $Y_{end}$ position – *Resets to **191**
Reads do not advance the clip position – Use **NR 28 (1Ch) D2:D3** to read the position. If need be, write to **NR 28 (1Ch):D1** to reset the clip index. When the clip window is enabled for sprites in **over border** mode, the **X** coordinates are internally doubled and the clip window origin is moved to the sprite origin inside the border

## NextREG 26 (1Ah) – Layer 0 (ULA/LoRes) Clip Window Definition

Data Bits

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Coordinate ($X_{beg}$,$X_{end}$,$Y_{beg}$,$Y_{end}$) | ■ | ■ | ● | ● | ● | ● | ● | ● | ● | ● | 8-bit value $X_{beg}$-$X_{end}$,$Y_{beg}$-$Y_{end}$ coordinate[1] | | * |

1 1st Write $X_{beg}$ position – *Resets to **0**
2nd Write $X_{end}$ position – *Resets to **255**
3rd Write $Y_{beg}$ position – *Resets to **0**
4th Write $Y_{end}$ position – *Resets to **191**
Reads do not advance the clip position – Use **NR 28 (1Ch) D4:D5** to read the position. If need be, write to **NR 28 (1Ch):D2** to reset the clip index
**NOTE:** LoRes may get a separate clip window in the future

## NextREG 27 (1Bh) – Layer 3 (Tilemap) Clip Window Definition

Data Bits

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Coordinate ($X_{beg}$,$X_{end}$,$Y_{beg}$,$Y_{end}$) | ■ | ■ | ● | ● | ● | ● | ● | ● | ● | ● | 8-bit value $X_{beg}$-$X_{end}$,$Y_{beg}$-$Y_{end}$ coordinate[1] | | * |

1 1st Write $X_{beg}$ position – *Resets to **0**
2nd Write $X_{end}$ position – *Resets to **159**
3rd Write $Y_{beg}$ position – *Resets to **0**
4th Write $Y_{end}$ position – *Resets to **255**
Reads do not advance the clip position – **NR 28 (1Ch) D6:D7** reads the position. If need be, write to **NR 28 (1Ch):D3** to reset the clip index and then do consecutive writes and reads to get to the value you're searching for. The **X** coordinates are internally doubled

## NextREG 28 (1Ch) – Clip Windows Control

Data Bits

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Layer 2 Clip Index | ■ | | | | | | | | ● | ● | Layer 2 Clip Index (0 – 3) | | |
| Sprites Layer Clip Index | ■ | | | | | | | ● | ● | | Sprites Clip Index (0 – 3) | | |
| Layer 0 / 1 Clip Index | ■ | | | | | | ● | ● | | | ULA / Timex Clip Index (0 – 3) | | |
| Layer 3 Clip Index | ■ | | | | | | ● | ● | | | Tilemap Clip Index (0 – 3) | | |
| Reserved | | ■ | 0 | 0 | 0 | 0 | | | | | Reserved, must be 0 | | |
| Layer 2 Clip Index Reset enable | | ■ | | | | | | | | ● | 1 to Reset the Layer 2 Clip Index | | |
| Sprites Clip Index Reset enable | | ■ | | | | | | | ● | | 1 to Reset the Sprites Clip Index | | |
| Layer 0/1 Clip Idx Reset enable | | ■ | | | | | | ● | | | 1 to Reset the ULA/Timex Clip Index | | |
| Layer 3 Clip Idx Reset enable | | ■ | | | | | ● | | | | 1 to Reset the Tilemap Clip Index | | |

**NOTE:** This NextREG may change in the future

## NextREG 30 (1Eh) – Active Video Line MSB

Data Bits

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Active Video Line (MSB) | ■ | | | | | | | | | ● | 1 for lines above 255 else 0 | | |
| Reserved | ■ | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | Reserved, must be 0 | | |

## NextREG 31 (1Fh) – Active Video Line LSB

Data Bits

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Active Video Line (LSB) | ■ | | ● | ● | ● | ● | ● | ● | ● | ● | Active Video Line (LSB) | | |

## NextREG 32 (20h) – Maskable Interrupt Generation

Data Bits

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| CTC | ■ | ■ | | | | | 0 | 0 | 0 | 0 | No CTC interrupt generated | | |
| | | | | | | | | | | ● | Generate CTC 0 interrupt | | |
| | | | | | | | | | ● | | Generate CTC 1 interrupt | | |
| | | | | | | | | ● | | | Generate CTC 2 interrupt | | |
| | | | | | | | ● | | | | Generate CTC 3 interrupt | | |
| Reserved | ■ | ■ | | | | | 0 | 0 | | | Reserved, must be 0 | | |
| ULA | ■ | ■ | | | | ● | | | | | Generate ULA interrupt | | |
| Line | ■ | ■ | ● | | | | | | | | Generate Line interrupt | | |

* Set bits on **R** indicate whether an interrupt occurred or is pending – alias of bits in **NextREGs 200 - 202 (C8h - CAh)**
* Set bits on **W** always generate a maskable interrupt ignoring enables – **NextREGs 196 - 198 (C4h - C6h)**

## NextREG 34 (22h) – Line Interrupt Control

Data Bits

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Line Interrupt Value MSB | ■ | ■ | | | | | | | | ● | MSB of line number | | 0 |
| Line Interrupt Control | ■ | ■ | | | | | | | ● | | Enable Line Interrupt* | | 0 |
| ULA Interrupt Control | ■ | ■ | | | | | | ● | | | Disable ULA Interrupt* | | 0 |
| Reserved | | ■ | 0 | 0 | 0 | 0 | 0 | | | | Reserved, must be 0 | | |
| | ■ | | | | | 0 | 0 | 0 | 0 | | Reserved, must be 0 | | |
| ULA Interrupt Signal | ■ | | | ● | | | | | | | ULA is currently asserting interrupt | | |

* Aliases of interrupt enable bits in **NextREG 196 (C4h)**

## NextREG 35 (23h) – Line Interrupt Value LSB

Data Bits

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Line Interrupt Value LSB | ■ | ■ | ● | ● | ● | ● | ● | ● | ● | ● | Lower 8-bits of line number | | 0 |

## NextREG 36 (24h) – Reserved

Protection against **OUT 3Bh, A** – See DISCiPLE disk interface for example.
In legacy modes set selected **NextREG** to **36 (24h)**

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| X Offset | ■ | ■ | ● | ● | ● | ● | ● | ● | ● | ● | 8-bit value of X Offset (0–255) | | 0 |

This setting refers to all ULA modes **except** Layer 1,0 – LoRes
**NextREG 104 (68h):D2** adds a half pixel to the scroll

## NextREG 39 (27h) – ULA Vertical Scroll Control

Data Bits

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Y Offset | ■ | ■ | ● | ● | ● | ● | ● | ● | ● | ● | 8-bit value of Y Offset (0–191) | | 0 |

This setting refers to all ULA modes **except** Layer 1,0 – LoRes

## NextREG 40 (28h) – Stored Palette Value and PS/2 Keymap Address MSB

Data Bits

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Stored Palette value | ■ | | ● | ● | ● | ● | ● | ● | ● | ● | See NextREG 68 (44h) | | |
| PS/2 Keymap Address MSB | ■ | | | | | | | | | ● | MSB of value in NR 41 (29h) | | |
| Keymap Selection | ■ | | 0 | | | | | | | | Select PS/2 Keymap | | 0 |
| | ■ | | 1 | | | | | | | | Select key joystick | | |
| Reserved | ■ | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Reserved, must be 0 | | |

## NextREG 41 (29h) – PS/2 Keymap Address LSB

Data Bits

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| PS/2 Keymap Address LSB | ■ | ■ | ● | ● | ● | ● | ● | ● | ● | ● | 8-bit value (0–255) | | 0 |

## NextREG 42 (2Ah) – PS/2 Keymap Data MSB

Data Bits

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| PS/2 Keymap Data MSB | ■ | | | | | | | | | ● | PS/2 Keymap Data MSB* | | |
| Reserved | ■ | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | Reserved, must be 0 | | |

* Not currently used by hardware. Write 0

## NextREG 43 (2Bh) – PS/2 Keymap Data LSB

Data Bits

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| PS/2 Keymap Data LSB | ■ | ■ | ● | ● | ● | ● | ● | ● | ● | ● | 8-bit value (0–255) | | |

A write causes the data to be written and auto-increments the Keymap Address

## NextREG 44 (2Ch) – DAC B Mirror (Left) / I²S Left Sample MSB

Data Bits

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| I²S Left Sample MSB | ■ | | ● | ● | ● | ● | ● | ● | ● | ● | 8-bit value (0–255) | | |
| 8-bit sample Left DAC B | | ■ | ● | ● | ● | ● | ● | ● | ● | ● | 8-bit value | | * |

* A soft reset sets a value of **128 (80h)**
The I²S Left Sample LSB is latched and can be read from **NextREG 45 (2Dh)** later

## NextREG 45 (2Dh) – DAC A+D Mirror (Mono) / I²S Sample LSB

Data Bits

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| I²S Last Sample LSB | ■ | | ● | ● | ● | ● | ● | ● | ● | ● | 8-bit value (0–255) | | |
| 8-bit sample DACs A + D | | ■ | ● | ● | ● | ● | ● | ● | ● | ● | 8-bit value | | * |

* A soft reset sets a value of **128 (80h)**
Returns the LSB of last sample read from **NextREG 44(2Ch)** or **NextREG 46(2Eh)**

## NextREG 46 (2Eh) – DAC C Mirror (Right) / I²S Right Sample MSB

Data Bits

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| I²S Right Sample MSB | ■ | | ● | ● | ● | ● | ● | ● | ● | ● | 8-bit value (0–255) | | |
| 8-bit sample Right DAC C | | ■ | ● | ● | ● | ● | ● | ● | ● | ● | 8-bit value | | * |

* A soft reset sets a value of **128 (80h)**
The I²S Right Sample LSB is latched and can be read from **NextREG 45 (2Dh)** later

## NextREG 47 (2Fh) – Layer 3 (Tilemap) Horizontal Scroll Control MSB

Data Bits

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Layer 3 X Scroll Offset MSB | ■ | ■ | | | | | | | ● | ● | Tilemap X Scroll Offset MSB | | 0 |
| Reserved | ■ | | 0 | 0 | 0 | 0 | 0 | 0 | | | Reserved, must be 0 | | |

Meaningful range: **0** to **319** in 40 tiles mode, **0** to **639** in 80 tiles mode

## NextREG 48 (30h) – Layer 3 (Tilemap) Horizontal Scroll Control LSB

Data Bits

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| X Offset LSB | ■ | ■ | ● | ● | ● | ● | ● | ● | ● | ● | 8-bit value (0–255) | | 0 |

Meaningful range: **0** to **319** in 40 tiles mode, **0** to **639** in 80 tiles mode

## NextREG 49 (31h) – Layer 3 (Tilemap) Vertical Scroll Control

Data Bits

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Y Offset | ■ | ■ | ● | ● | ● | ● | ● | ● | ● | ● | 8-bit value (0–255) | | 0 |

## NextREG 50 (32h) – Layer 1,0 (LoRes) Horizontal Scroll Control

Data Bits

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| X Offset | ■ | ■ | ● | ● | ● | ● | ● | ● | ● | ● | 8-bit value of X Offset (0–255) | | 0 |

Layer 1,0 (LoRes) scrolls in half-pixels at the same resolution and smoothness as Layer 2

## NextREG 51 (33h) – Layer 1,0 (LoRes) Vertical Scroll Control

Data Bits

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Y Offset | ■ | ■ | ● | ● | ● | ● | ● | ● | ● | ● | 8-bit value of Y Offset (0–191) | | 0 |

Layer 1,0 (LoRes) scrolls in half-pixels at the same resolution and smoothness as Layer 2

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| When **NR09:D4** is set | ■ | ■ | ● | ● | ● | ● | ● | ● | ● | ● | Sprite number[1] / Pattern Number[2] | | |
| | | | ● | | | | | | | | Pattern Address Offset[3] | | |

The above applies only when the sprites port is in Lockstep and effectively performs an **OUT** to port **12347** (**303Bh**) with the same value otherwise the section below applies

| When **NR09:D4** is **NOT** set | ■ | ■ | 0 | ● | ● | ● | ● | ● | ● | ● | Sprite number[1] | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

1 Values are **0** to **127**
2 Values are **0** to **63**
3 Adds **128** to pattern address
This register selects which sprite has its attributes connected to the registers that follow: **NextREG 53** (**36h**) – **NextREG 57** (**39h**) and their auto-incremented counterparts **NextREG 117** (**75h**) – **NextREG 121** (**79h**)

## NextREG 53 (35h) – Sprite Attribute 0

Data Bits

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| X Coordinate LSB | ■ | ● | ● | ● | ● | ● | ● | ● | ● | ● | Sprite X Coordinate LSB | | |

MSB is in **NextREG 55** (**37h**):**D0**

## NextREG 54 (36h) – Sprite Attribute 1

Data Bits

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Y Coordinate LSB | ■ | ● | ● | ● | ● | ● | ● | ● | ● | ● | Sprite Y Coordinate LSB | | |

MSB is in **NextREG 57** (**39h**):**D0**

## NextREG 55 (37h) – Sprite Attribute 2

Data Bits

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Sprite Attribute 2 | ■ | ■ | e | e | e | e | d | c | b | a | See notes | | |

a For **relative** sprites: Indicates that **e** is relative to the anchor's palette offset
For normal sprites: Sprite's **X Coordinate MSB** (See **NextREG 53** (**35h**) for LSB)
b 90° Clockwise Rotation Control (**0** = **No**, **1** = **Yes**)
c Vertical Mirror Control (**0** = **No**, **1** = **Yes**)
d Horizontal Mirror Control (**0** = **No**, **1** = **Yes**)
e 4-bit palette offset
Rotation is applied before mirroring

## NextREG 56 (38h) – Sprite Attribute 3

Data Bits

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Sprite Attribute 3 | ■ | ■ | c | b | a | a | a | a | a | a | See notes | | |

a Sprite pattern to use. Possible values = **0** to **63**
b Attribute 4 switch (**0** = **No**, **1** = **Yes**)
If **b** = **0** then the sprite is fully described by **Attributes 0** to **3**. The sprite pattern is an 8-bit one identified by pattern **a** and is an anchor and cannot be made relative. Sprite display behaves as if **Attribute 4** = **0**
If **b** = **1** then the sprite is further described by **Attribute 4** that follows in **NextREG 57** (**39h**)
c Visibility Control (**0** = **Invisible**, **1** = **Visible**)

## NextREG 57 (39h) – Sprite Attribute 4

Data Bits

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Sprite Attribute 4 | ■ | ■ | f | e | d | c | c | b | b | a | See notes | | |

a For **relative** sprites: Indicates that the sprite pattern number is relative to the anchor's
For **normal** sprites: Sprite's **Y Coordinate MSB** (See **NextREG 54** (**36h**) for LSB)
b For **normal** and **relative, composite type** sprites indicates **X** direction Magnification:
(**00** = **1x**, **01** = **2x**, **10** = **4x**, **11** = **8x**)
For relative, unified type sprites it's **0**
c For **normal** and **relative, composite type** sprites indicates **Y** direction Magnification:
(**00** = **1x**, **01** = **2x**, **10** = **4x**, **11** = **8x**)
For **relative, unified type** sprites it's **0**
d For **normal** sprites, indicates that the attached relative sprites are: **0** for **Composite**, **1** for **Unified**
For **relative** sprites contains the 7th pattern bit if the sprite pattern is 4-bit.
e For **normal** sprites contains the 7th pattern bit if the sprite pattern is 4-bit.
For **relative** sprites it's **1**.
f 4-bit pattern switch: **1** if the sprite pattern is 4-bit otherwise **0**
{**f**,**e**} must not equal {**0**,**1**} as this combination is used to indicate a relative sprite. See notes above

## NextREG 64 (40h) – Palette Index Select

Data Bits

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Palette Index Select | ■ | ■ | ● | ● | ● | ● | ● | ● | ● | ● | Palette index number | | |

Selects the palette index to change the associated colour
For ULA only, **INK**s are mapped to indices **0** through **7**, **BRIGHT INK**s to indices **8** through **15**, **PAPER**s to indices **16** through **23** and **BRIGHT PAPER**s to indices **24** through **31**
In EnhancedULA mode, **INK**s come from a subset of indices from **0** through **127** and **PAPER**s from a subset of indices from **128** through **255**.
The number of active indices depends on the number of attribute bits assigned to **INK** and **PAPER** out of the attribute byte
In ULAplus mode, the last **64** entries (indices **192** to **255**) hold the ULAplus palette
The ULA always takes border colour from **PAPER** for standard ULA and EnhancedULA

## NextREG 65 (41h) – 8-bit Palette Data

Data Bits

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 8-bit Palette Entry | ■ | ■ | ● | ● | ● | ● | ● | ● | ● | ● | Colour entry in RRRGGGBB format | | |

The lower blue bit of the 9-bit internal colour will be the logical OR of Bits **0** and **1** of the 8-bit entry. After each write, the palette index is auto-incremented to the next index if the auto-increment has been enabled in **NextREG 67** (**43h**):**D7**
Reads do not auto-increment the index. Any other bits associated with the index will be zeroed

## NextREG 66 (42h) – EnhancedULA Attribute Byte Format

Data Bits

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Attribute Byte Format | ■ | ■ | ● | ● | ● | ● | ● | ● | ● | ● | Attrbute byte's INK representation mask | | * |

* Soft reset defaults to **7**
Not set bits, indicate **PAPER**, Acceptable values are made by setting each bit from **0** on, in sequence: (**1**,**3**,**7**,**15**,**31**,**63**,**127** and **255**) which effectively splits the attribute byte setting the **INK**s from the right side and the **PAPER**s from what's left. **INK**s are mapped from Index **0** onwards on the palette while **PAPER**s and **BORDER** are mapped from Index **128** onwards

taken from the fallback colour defined in **NextREG 74** (**4Ah**)
If the mask is not one of those listed above, the **INK** is still the result of logically **AND**ing the mask with the attribute byte but the **PAPER** and **BORDER** will be taken from the fallback colour
Example:
**00111011** will be ANDed with the attribute byte to form the **INK** index and **PAPER** will come from the fallback colour

## NextREG 67 (43h) – Palette Control

Data Bits

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| EnhancedULA control | ■ | ■ | | | | | | | | ● | Enable EnhancedULA | |
| Active ULA[1] Palette | ■ | ■ | | | | | | | 0 | | First Palette | |
| | | | | | | | | | 1 | | Second Palette | |
| Active Layer 2 Palette | ■ | ■ | | | | | | 0 | | | First Palette | |
| | | | | | | | | 1 | | | Second Palette | |
| Active Sprites Palette | ■ | ■ | | | | | 0 | | | | First Palette | |
| | | | | | | | 1 | | | | Second Palette | |
| Palette Select for Read/Write | ■ | ■ | | | 0 | 0 | 0 | | | | Layers 0/1 (ULA) First | |
| | | | | | 1 | 0 | 0 | | | | Layers 0/1 (ULA) Second | |
| | | | | | 0 | 0 | 1 | | | | Layer 2 First | |
| | | | | | 1 | 0 | 1 | | | | Layer 2 Second | |
| | | | | | 0 | 1 | 0 | | | | Sprites First | |
| | | | | | 1 | 1 | 0 | | | | Sprites Second | |
| | | | | | 0 | 1 | 1 | | | | Layer 3 (Tilemap) First | |
| | | | | | 1 | 1 | 1 | | | | Layer 3 (Tilemap) Second | |
| Palette Auto-increment Control | ■ | ■ | ● | | | | | | | | Disable Palette Write Auto-increment | |

* After a soft reset defaults to **000**
[1] ULA refers to all ULA modes (Layers 0 and 1)

## NextREG 68 (44h) – 9-bit Palette Data

Data Bits

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| MSB Colour (1st Write) Non L2 | ■ | ● | ● | ● | ● | ● | ● | ● | ● | ● | MSB (RRRGGGBB) format – non L2 palette | |
| LSB Blue (2nd Write) Non L2 | ■ | ■ | | | | | | | | ● | LSB B format – non L2 palette | |
| Reserved Non L2 | ■ | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Reserved, must be 0 – non L2 palette | |
| MSB Colour (1st Write) L2 | ■ | ● | ● | ● | ● | ● | ● | ● | ● | ● | MSB (RRRGGGBB) format – L2 palette | |
| LSB Blue + Priority L2 (2nd W) | ■ | ■ | ● | | | | | | | ● | L2 priority (D7) and LSB B(D0) | |
| Reserved L2 | ■ | | | 0 | 0 | 0 | 0 | 0 | 0 | | Reserved, must be 0 L2 palette | |

9-bit Palette Data is entered in two consecutive writes; the second write auto-increments the palette index if auto-increment is enabled in **NextREG 67** (**43h**):**D7**
If writing an L2 palette, the second write's **D7** holds the **L2 priority** bit which if set (**1**) brings the colour defined at that index on top of all other layers. If you also need the same colour in regular priority (for example: for environmental masking) you will have to set it up again, this time with no priority
Reads return the second byte and do not auto-increment
Writes to **NextREG**s **64**, **65** and **67** (**40h**, **41h** and **43h**) reset to the 1st write

## NextREG 74 (4Ah) – Fallback Colour Value

Data Bits

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Fallback Colour | ■ | ■ | ● | ● | ● | ● | ● | ● | ● | ● | 8-bit colour if all layers are transparent | |

* Soft reset sets the default fallback to **227** (**E3h**) as it must be the same for when ULAplus programs hit the transparent colour, otherwise nothing will be displayed

## NextREG 75 (4Bh) – Sprite Transparency Index

Data Bits

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Transparency Index | ■ | ■ | ● | ● | ● | ● | ● | ● | ● | ● | Sprite colour index treated as transparent | |

* Soft reset defaults to **227** (**E3h**)
For 4-bit sprites, only 4-bits are used (from **D0** to **D3**)
For example for 8-bit transparency index **227** (**E3h**) the 4-bit equivalent will be **03** (**03h**)

## NextREG 76 (4Ch) – Layer 3 (Tilemap) Transparency Index

Data Bits

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Transparency Index | ■ | ■ | | | | | ● | ● | ● | ● | 4-bit index treated as transparent | |
| Reserved | ■ | ■ | 0 | 0 | 0 | 0 | | | | | Reserved, must be 0 | |

* Soft reset defaults to **15** (**0Fh**)

## NextREG 80 (50h) – MMU Slot 0 Control

Data Bits

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| MMU Slot 0 Control | ■ | ■ | ● | ● | ● | ● | ● | ● | ● | ● | 8K RAM page for address 0000h–1FFFh | |

* Default **255** (**FFh**)
Pages range from **0** to **223** on a fully expanded Next. A value of **255** (**FFh**) makes the ROM become visible

## NextREG 81 (51h) – MMU Slot 1 Control

Data Bits

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| MMU Slot 1 Control | ■ | ■ | ● | ● | ● | ● | ● | ● | ● | ● | 8K RAM page for address 2000h–3FFFh | |

* Default **255** (**FFh**)
Pages range from **0** to **223** on a fully expanded Next. A value of **255** (**FFh**) makes the ROM become visible

## NextREG 82 (52h) – MMU Slot 2 Control

Data Bits

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| MMU Slot 2 Control | ■ | ■ | ● | ● | ● | ● | ● | ● | ● | ● | 8K RAM page for address 4000h–5FFFh | |

* Default **10** (**0Ah**)
Pages range from **0** to **223** on a fully expanded Next

### NextREG 83 (53h) – MMU Slot 3 Control

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| MMU Slot 3 Control | ■ | ■ | ● | ● | ● | ● | ● | ● | ● | ● | 8K RAM page for address 6000h–7FFFh | | * |

* Default **11** (**0Bh**) – Pages range from **0** to **223** on a fully expanded Next

### NextREG 84 (54h) – MMU Slot 4 Control

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| MMU Slot 4 Control | ■ | ■ | ● | ● | ● | ● | ● | ● | ● | ● | 8K RAM page for address 8000h–9FFFh | | * |

* Default **4** (**04h**) – Pages range from **0** to **223** on a fully expanded Next

### NextREG 85 (55h) – MMU Slot 5 Control

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| MMU Slot 5 Control | ■ | ■ | ● | ● | ● | ● | ● | ● | ● | ● | 8K RAM page for address A000h–BFFFh | | * |

* Default **5** (**05h**) – Pages range from **0** to **223** on a fully expanded Next

### NextREG 86 (56h) – MMU Slot 6 Control

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| MMU Slot 6 Control | ■ | ■ | ● | ● | ● | ● | ● | ● | ● | ● | 8K RAM page for address C000h–DFFFh | | * |

* Default **0** (**00h**) – Pages range from **0** to **223** on a fully expanded Next

### NextREG 87 (57h) – MMU Slot 7 Control

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| MMU Slot 7 Control | ■ | ■ | ● | ● | ● | ● | ● | ● | ● | ● | 8K RAM page for address E000h–FFFFh | | * |

* Default **1** (**01h**) – Pages range from **0** to **223** on a fully expanded Next

### NextREG 96 (60h) – Copper Data 8-bit Write

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Copper Instruction 8-bit | | ■ | ● | ● | ● | ● | ● | ● | ● | ● | Byte to write to copper instruction memory | | |

Each Copper Instruction is two-bytes long. After a write, the Copper address is auto-incremented to the next memory position

### NextREG 97 (61h) – Copper Address LSB

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Copper memory address LSB | ■ | ■ | ● | ● | ● | ● | ● | ● | ● | ● | Copper instruction memory address (LSB) | | 0 |

Copper memory addresses range over **0** through **2047** (**7FFh**)

### NextREG 98 (62h) – Copper Control

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Copper Memory Address MSB | ■ | ■ | | | | | | ● | ● | ● | Copper Instruction Memory Address (MSB) | | 0 |
| Copper Start Control | ■ | ■ | 0 | 0 | | | | | | | Copper fully stopped | | * |
| | | | 0 | 1 | | | | | | | Copper start, exec from 0, loop | | |
| | | | 1 | 0 | | | | | | | Copper start, exec from last, loop | | |
| | | | 1 | 1 | | | | | | | Copper start, exec from 0, rest. at raster 0,0 | | |

* Soft reset defaults to **000**
Copper memory addresses range from **0** through **2047** (**7FFh**)
Note: Writing the same copper start control value does not reset the copper

### NextREG 99 (63h) – Copper Data 16-bit Write

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Copper data | | ■ | ● | ● | ● | ● | ● | ● | ● | ● | 1st write MSB, 2nd write LSB | | 0 |

The 16-bit value is written in pairs. The first 8-bits are the MSB and are destined for an even copper instruction address. The second 8-bits are the LSB and are destined for an odd copper instruction address
After each write, the copper address is auto-incremented to the next memory position.
After a write to an odd address, the entire 16-bits is written to copper memory at once

### NextREG 100 (64h) – Vertical Line Count Offset

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Line counter offset | | ■ | ● | ● | ● | ● | ● | ● | ● | ● | Offset added to the vertical line counter* | | |

* Affects Copper, line interrupt and active line count
Normally the ULA's pixel row **0**, aligns with vertical line count **0**. With a non-zero offset, the ULA's pixel row **0** will align with the vertical line offset. For example if the offset is **32** then vertical line **32** will correspond to the first pixel row in the ULA and vertical line **0** will align with the first pixel row of *Layer 3* (Tilemap) and the *Sprite Layer*
**NOTE:** Since a change in offset takes effect when the ULA reaches row **0**, the change can take up to one frame to occur

### NextREG 104 (68h) – ULA Control

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Stencil Mode control | ■ | ■ | | | | | | | | ● | Enable Stencil Mode[1] | | 0 |
| Reserved | | ■ | | | | | | | 0 | | Reserved, must be 0 | | 0 |
| ULA Half-Pixel Scroll | ■ | ■ | | | | | | ● | | | ULA Half-Pixel Scroll enabled[2] | | 0 |
| ULAplus Control | ■ | ■ | | | | | ● | | | | ULAplus Enabled | | 0 |
| Extended keys in 8x5 matrix | ■ | ■ | | | | ● | | | | | Cancel entries in matrix for extended keys | | 0 |
| Blend Colour Used in SLU modes 6 & 7 | ■ | ■ | | | 0 | 0 | | | | | ULA Colour | | 0 |
| | | | | | 1 | 0 | | | | | ULA + Tilemap mix | | |
| | | | | | 1 | 1 | | | | | Tilemap Colour | | |
| | | | | | 0 | 1 | | | | | No Colour Blending | | |
| Output Control | | ■ | ● | | | | | | | | Disable ULA output | | 0 |

1 When both the ULA and Layer 3 (Tilemap) are enabled, if either are transparent, the result is transparent otherwise the result is a logical AND of both colours
2 Setting may change

### NextREG 105 (69h) – Display Control 1

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Port 255 (FFh) "Timex" alias | ■ | ■ | | | ● | ● | ● | ● | ● | ● | Port 255 (FFh) alias | | |
| Port 32765 (7FFDh):D3 alias | ■ | ■ | | ● | | | | | | | ULA Shadow Display Enable | | |
| Port 4667 (123Bh):D1 alias | ■ | ■ | ● | | | | | | | | Layer 2 Enable | | |

### NextREG 106 (6Ah) – Layer 1,0 (LoRes) Control

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| LoRes \| ULAplus palette Offset | ■ | ■ | | | | | ● | ● | ● | ● | LoRes Palette Offset | | 0 |
| | | | | | | | | | ● | ● | ULAplus Palette Offset | | 0 |
| Radastan Mode DFILE Source | ■ | ■ | | | | ● | | | | | Timex DFILE source[1] | | 0 |
| LoRes mode | ■ | ■ | | | ● | | | | | | Radastan Mode Enable | | 0 |
| Reserved | ■ | ■ | 0 | 0 | | | | | | | Reserved, must be 0 | | |

1 Layer 1,0 (LoRes) occupies both Timex display files but Radastan mode has only half the colour depth thus occupies only one DFILE – at either **16384** (**4000h**) or **24576** (**6000h**). When this bit is set, the source DFILE used is the opposite of the ULA so that Radastan can co-exist with the normal Layer 0 (ULA) screen

### NextREG 107 (6Bh) – Layer 3 (Tilemap) Control

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Layer 3 Priority | ■ | ■ | | | | | | | | ● | Tilemap on top of ULA Enable | | 0 |
| 512 Tile mode Control | ■ | ■ | | | | | | | ● | | Activate 512 Tile mode[1] | | 0 |
| Reserved | ■ | ■ | | | | | | | 0 | | Reserved, must be 0 | | 0 |
| Text mode Control | ■ | ■ | | | | | | ● | | | Text mode Enable | | 0 |
| Layer 3 palette Select | ■ | ■ | | | | | 0 | | | | Palette 0 | | 0 |
| | | | | | | | 1 | | | | Palette 1 | | |
| Attribute Entry Control | ■ | ■ | | | | ● | | | | | Attribute entry Disable[2] | | 0 |
| Layer 3 Size Control | ■ | ■ | | | | | 0 | | | | 40 x 32 | | 0 |
| | | | | | | | 1 | | | | 80 x 32 | | |
| Layer 3 Control | ■ | ■ | ● | | | | | | | | Tilemap Enable | | 0 |

1 If this bit is set, **NextREG 108 (6Ch):D0** changes meaning
2 If this bit is set then the Layer 3 tilemap entries are only a single byte **Tile ID** and the attribute byte comes from **NextREG 108 (6Ch)** instead

### NextREG 108 (6Ch) – Default Layer 3 (Tilemap) Attribute*

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| If 512 Tile mode is disabled | ■ | ■ | | | | | | | | ● | ULA over Layer 3 | | 0 |
| If 512 Tile mode is enabled | ■ | ■ | | | | | | | | ● | Bit 8 of the tile number | | |
| Rotate 90° Control | ■ | ■ | | | | | | | ● | | Rotate | | 0 |
| Y Mirror Control | ■ | ■ | | | | | | ● | | | Y Mirror | | 0 |
| X Mirror Control | ■ | ■ | | | | | ● | | | | X Mirror | | 0 |
| Palette Offset | ■ | ■ | ● | ● | ● | ● | ● | | | | Palette Offset | | 0 |

* Active if **NextREG 107** (**6Bh**):**D5** is set. If **NextREG 107** (**6Bh**):**D3** is set, the palette offset is extended to **D7**:**D1** replacing the rotate and mirror bits

### NextREG 110 (6Eh) – Layer 3 (Tilemap) Base Address

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| MSB of Layer 3 base address in Banks 5 & 7 | ■ | ■ | | 0 | ● | ● | ● | ● | ● | ● | Address in bank 5 or bank 7 – Entered | | * |
| | ■ | ■ | ● | | | | | | | | together with bit 7 – bit 6 is 0 | | |

* Soft Reset value is **108** (**6Ch**) read as **44** (**2Ch**)
Writes are the MSB of an address in bank 5 **16384 - 32767** (**4000h – 7FFFh**) or in bank 7 **49152 - 65535** (**C000h – FFFFh**) with **D6** ignored and always read back as 0. This means tile definitions start on 256-byte boundaries
Bank 7 addresses wrap around its 8K boundary

### NextREG 111 (6Fh) – Layer 3 (Tilemap) - Tile Definitions Base Address

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| MSB of Layer 3 Tile definitions base address in Banks 5 & 7 | ■ | ■ | | 0 | ● | ● | ● | ● | ● | ● | Address in bank 5 or bank 7 – Entered | | * |
| | ■ | ■ | ● | | | | | | | | together with bit 7 – bit 6 is 0 | | |

* Soft Reset value is **76** (**4Ch**) read as **12** (**0Ch**)
Writes are the MSB of an address in bank 5 **16384 - 32767** (**4000h – 7FFFh**) or in bank 7 **49152 - 65535** (**C000h – FFFFh**) with **D6** ignored and always read back as 0. This means tile definitions start on 256-byte boundaries
Bank 7 addresses wrap around its 8K boundary

### NextREG 112 (6Ah) – Layer 2 Resolution Control*

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Layer 2 Palette Offset | ■ | ■ | | | | | ● | ● | ● | ● | Layer 2 4-bit Palette Offset | | 0 |
| Layer 2 Resolution Select** | ■ | ■ | | | 0 | 0 | | | | | 256 x 192 x 8bpp (Layer 2 standard) | | 0 |
| | | | | | 0 | 1 | | | | | 320 x 256 x 8bpp (Layer 2, Mode 2) | | |
| | | | | | 1 | 0 | | | | | 640 x 256 x 4bpp (Layer 2, Mode 3) | | |
| Reserved | ■ | ■ | 0 | 0 | | | | | | | Reserved, must be 0 | | |

* The *Layer 2* screen occupies **48K** or **80K**, which is stored in **3** consecutive **16 KB** banks for **256 x 192 x 8bpp** or **5** consecutive banks for **320 x 256** and **640 x 256** resolutions, When in *NextZXOS*/*NextBASIC*, banks **9 - 11** are used for the visible and shadow *Layer 2*(.0) screen (after power-on, the hardware default to banks **8 - 10** for displayed and **11 - 13** for shadow screen but this gets modified by *NextZXOS* – See *Chapter 23*), These can be set using *Layer 2* RAM Page **NextREG 18** (**12h**) and *Layer 2* RAM Shadow Page **NextREG 19** (**13h**) (avoid banks **5**, **7** and **8** to be used as *Layer 2* screen, unless you are familiar with SRAM and BRAM of the board and how the ULA screen memory has special treatment in Next's FPGA)
** Each pixel of *Layer 2* is assigned 1 byte (8-bits) (in **8bpp** modes) or 1 nibble (4-bits) (in **640 x 256 x 4bpp** mode) of video memory. This means *Layer 2* consumes a total of either **48K** or **80K** of memory. The memory is divided in **16K** "*strips*" filling one standard Spectrum bank, In **256x192** mode the strips are horizontal so that **16K** represent **64** vertical lines on the display, In **320x256** and **640x256** modes the strips are vertical so that each **16K** corresponds to **64** or **128** columns respectively – See *Chapters 15* and *16*

### NextREG 113 (71h) – Layer 2 Horizontal Scroll Control MSB

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| X Coordinate MSB | ■ | ■ | | | | | | | | ● | Layer 2 X Coordinate MSB | | |
| Reserved | ■ | ■ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | Reserved, must be 0 | | |

LSB is in **NextREG 22** (**16h**)

### NextREG 117 (75h) – Sprite Attribute 0 (Auto-incrementing)

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| X Coordinate LSB | ■ | ● | ● | ● | ● | ● | ● | ● | ● | ● | Sprite X Coordinate LSB | | |

Same as **NextREG 53** (**35h**) but writes also auto-increment the sprite number

### NextREG 118 (76h) – Sprite Attribute 1 (Auto-incrementing)

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Y Coordinate LSB | ■ | ● | ● | ● | ● | ● | ● | ● | ● | ● | Sprite Y Coordinate LSB | | |

Same as **NextREG 54** (**36h**) but writes also auto-increment the sprite number

### NextREG 119 (77h) – Sprite Attribute 2 (Auto-incrementing)

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Sprite Attribute 2 | ■ | e | e | e | e | d | c | b | a | | See notes | | |

**a,b,c,d,e:** Same as **NextREG 55** (**37h**) but writes also auto-increment the sprite number

### NextREG 120 (78h) – Sprite Attribute 3 (Auto-incrementing)

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Sprite Attribute 3 | ■ | c | b | a | a | a | a | a | a | | See notes | | |

**a,b,c:** Same as **NextREG 56** (**38h**) but writes also auto-increment the sprite number

### NextREG 121 (79h) – Sprite Attribute 4 (Auto-incrementing)

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Sprite Attribute 4 | ■ | f | e | d | c | c | b | b | a | | See notes | | |

**a,b,c,d,e,f:** Same as **NextREG 57** (**39h**) but writes also auto-increment the sprite number

### NextREG 127 (7Fh) – User Register 0

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| User Register | ■ | ■ | ● | ● | ● | ● | ● | ● | ● | ● | User Register | | * |

* Soft reset defaults to **255** (**FFh**)

**CAUTION: NextREG numbers above 127 (7Fh) are inaccessible to the Copper**

### NextREG 128 (80h) – Expansion Bus Enable

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | **AFTER SOFT RESET** (Copied into bits D4 through D7) | | |
| Memory Cycles and $\overline{\text{ROMCS}}$ | ■ | ■ | | | | | | | | ● | Memory cycles Disable/ Ignore $\overline{\text{ROMCS}}$ | ■ | 0 |
| I/O Cycles and $\overline{\text{IORQULA}}$ | ■ | ■ | | | | | | | ● | | I/O cycles Disable / Ignore $\overline{\text{IORQULA}}$ | ■ | 0 |
| divMMC $\overline{\text{ROMCS}}$ ROM Enable | ■ | ■ | | | | | | ● | | | ROM from divMMC banks 14/15 Enable | ■ | 0 |
| Expansion bus Enable | ■ | ■ | | | | | ● | | | | Expansion bus Enable | ■ | 0 |
| | | | | | | | | | | | **IMMEDIATE** | | |
| Memory Cycles and $\overline{\text{ROMCS}}$ | ■ | ■ | | | | ● | | | | | Memory cycles Disable/ Ignore $\overline{\text{ROMCS}}$ | ■ | 0 |
| I/O Cycles and $\overline{\text{IORQULA}}$ | ■ | ■ | | | ● | | | | | | I/O cycles Disable / Ignore $\overline{\text{IORQULA}}$ | ■ | 0 |
| divMMC $\overline{\text{ROMCS}}$ ROM Enable | ■ | ■ | | ● | | | | | | | ROM from divMMC banks 14/15 Enable | ■ | 0 |
| Expansion bus Enable | ■ | ■ | ● | | | | | | | | Expansion bus Enable | ■ | 0 |

### NextREG 129 (81h) – Expansion Bus Control

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| MAX CPU Speed when Expansion Bus is enabled[1] | ■ | ■ | | | | | | | 0 | 0 | 3.5 MHz | ■ | * |
| | | | | | | | | | 0 | 1 | 7 MHz | | |
| | | | | | | | | | 1 | 0 | 14 MHz | | |
| | | | | | | | | | 1 | 1 | 28 MHz | | |
| Propagate MAX CPU Clock ena. | ■ | ■ | | | | ● | | | | | Propagate MAX CPU Clock at all times[2] | ■ | 0 |
| Disable Exp.Bus NMI Debounce | ■ | ■ | | | ● | | | | | | Disables the NMI debounce[3] | ■ | 0 |
| ULA override control | ■ | ■ | | ● | | | | | | | Allows peripherals to override ULA[4] | ■ | 0 |
| Exp. bus $\overline{\text{ROMCS}}$ state flag | ■ | ● | | | | | | | | | $\overline{\text{ROMCS}}$ asserted on Expansion Bus | ■ | 0 |

* Hard reset defaults to **00**
1 Currently fixed at **00**
2 Applies even when the Expansion Bus is disabled
3 A value of **1** disables the Expansion BSus' NMI debounce as required by some peripherals like the Opus Discovery
4 A value of **1** allows peripherals to override the ULA on even port reads as equired by some peripherals like the Rotronics Wafadrive

### NextREG 130 (82h) – Internal Port Decoding Control 1/4

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Enable Timex | ■ | ■ | | | | | | | | ● | Port FFh | | 1 |
| Enable Paging | ■ | ■ | | | | | | | ● | | Port 7FFDh | | 1 |
| Enable Next Memory Paging | ■ | ■ | | | | | | ● | | | Port DFFDh | | 1 |
| Enable +3 Paging | ■ | ■ | | | | | ● | | | | Port 1FFDh | | 1 |
| Enable +3 Floating Bus | ■ | ■ | | | | ● | | | | | +3 Floating bus | | 1 |
| Enable DMA | ■ | ■ | | | ● | | | | | | Port 6Bh (DMA) | | 1 |
| Enable Kempston Port 1 | ■ | ■ | | ● | | | | | | | Port 1Fh (Kempston / MD 1) | | 1 |
| Enable Kempston Port 2 | ■ | ■ | ● | | | | | | | | Port 37h (Kempston / MD 2) | | 1 |

### NextREG 131 (83h) – Internal Port Decoding Control 2/4

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Enable divMMC | ■ | ■ | | | | | | | | ● | Port E3h (divMMC Control) | | 1 |
| Enable Multiface | ■ | ■ | | | | | | | ● | | Multiface (two variable ports) | | 1 |
| Enable I²C | ■ | ■ | | | | | | ● | | | Ports 103Bh, 113Bh (I²C) | | 1 |
| Enable SPI | ■ | ■ | | | | | ● | | | | Ports E7h, EBh (SPI) | | 1 |
| Enable UART | ■ | ■ | | | | ● | | | | | Ports 133Bh, 143Bh, 153Bh,163Bh (UART) | | 1 |
| Enable Kempston Mouse | ■ | ■ | | | ● | | | | | | Ports FADFh,FBDFh,FFDFh mouse* | | 1 |
| Enable Sprites | ■ | ■ | | ● | | | | | | | Ports 57h,5Bh,303Bh (Sprites) | | 1 |
| Enable Layer 2 | ■ | ■ | ● | | | | | | | | Port 123Bh (Layer2) | | 1 |

* Also disables Kempston alias on port DFh

### NextREG 132 (84h) – Internal Port Decoding Control 3/4

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Enable AY | ■ | ■ | | | | | | | | ● | Ports FFFDh, BFFDh (AY) | | 1 |
| Enable Soundrive DAC Mode 1 | ■ | ■ | | | | | | | ● | | Ports 0Fh,1Fh, 4Fh, 5Fh (DAC SD1) | | 1 |
| Enable Soundrive DAC Mode 2 | ■ | ■ | | | | | | ● | | | Ports F1h, F3h, F9h ,FBh (DAC SD2) | | 1 |
| Enable Profi/Covox Stereo DAC | ■ | ■ | | | | | ● | | | | Ports 3Fh, 5Fh (DAC stereo-Profi/Covox) | | 1 |
| Enable Covox Stereo DAC | ■ | ■ | | | | ● | | | | | Ports 0Fh, 4Fh (DAC stereo-Covox) | | 1 |
| Enable Pentagon/ATM DAC | ■ | ■ | | | ● | | | | | | Port FBh (DAC mono-Pentagon) (SD2 off) | | 1 |
| Enable Covox/GS Mono DAC | ■ | ■ | | ● | | | | | | | Port B3h (DAC mono-GS/Covox) | | 1 |
| Enable SPECdrum Mono DAC | ■ | ■ | ● | | | | | | | | Port DFh (DAC mono-SPECdrum)* | | 1 |

* Also a Kempston port 1Fh alias

### NextREG 133 (85h) – Internal Port Decoding Control 4/4 (MSB)

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Enable ULAplus | ■ | ■ | | | | | | | | ● | Ports BF3Bh, FF3Bh (ULAplus) | ■ | 1 |
| Enable DMA | ■ | ■ | | | | | | | ● | | Port 0Bh (Z80DMA) | ■ | 1 |
| Enable Pentagon 1024 Memory | ■ | ■ | | | | | | ● | | | Port EFF7h (Pentagon 1024 memory) | ■ | 0 |
| Enable CTC | ■ | ■ | | | | | ● | | | | Ports 183Bh - 1F3Bh (CTC Channels 0-7) | ■ | 1 |
| Register Reset | ■ | ■ | ● | | | | | | | | Register reset mode (soft or hard reset sel.) | ■ | 1 |

### NextREG 134 (86h) – Expansion Bus Port Decoding Control 1/4

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Enable Timex | ■ | ■ | | | | | | | | ● | Port FFh | ■ | 1 |
| Enable Paging | ■ | ■ | | | | | | | ● | | Port 7FFDh | ■ | 1 |
| Enable Next Memory Paging | ■ | ■ | | | | | | ● | | | Port DFFDh | ■ | 1 |
| Enable +3 Paging | ■ | ■ | | | | | ● | | | | Port 1FFDh | ■ | 1 |
| Enable +3 Floating Bus | ■ | ■ | | | | ● | | | | | +3 Floating bus | ■ | 1 |
| Enable DMA | ■ | ■ | | | ● | | | | | | Port 6Bh (DMA) | ■ | 1 |
| Enable Kempston Port 1 | ■ | ■ | | ● | | | | | | | Port 1Fh (Kempston / MD 1) | ■ | 1 |
| Enable Kempston Port 2 | ■ | ■ | ● | | | | | | | | Port 37h (Kempston / MD 2) | ■ | 1 |

### NextREG 135 (87h) – Expansion Bus Port Decoding Control 2/4

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Enable divMMC | ■ | ■ | | | | | | | | ● | Port E3h (divMMC Control) | ■ | 1 |
| Enable Multiface | ■ | ■ | | | | | | | ● | | Multiface (two variable ports) | ■ | 1 |
| Enable I²C | ■ | ■ | | | | | | ● | | | Ports 103Bh, 113Bh (I²C) | ■ | 1 |
| Enable SPI | ■ | ■ | | | | | ● | | | | Ports E7h, EBh (SPI) | ■ | 1 |
| Enable UART | ■ | ■ | | | | ● | | | | | Ports 133Bh, 143Bh, 153Bh, 163Bh(UART) | ■ | 1 |
| Enable Kempston Mouse | ■ | ■ | | | ● | | | | | | Ports FADFh,FBDFh,FFDFh mouse[1] | ■ | 1 |
| Enable Sprites | ■ | ■ | | ● | | | | | | | Ports 57h,5Bh,303Bh (Sprites) | ■ | 1 |
| Enable Layer 2 | ■ | ■ | ● | | | | | | | | Port 123Bh (Layer2) | ■ | 1 |

1 Also disables Kempston alias on port DFh

### NextREG 136 (88h) – Expansion Bus Port Decoding Control 3/4

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Enable AY | ■ | ■ | | | | | | | | ● | Ports FFFDh, BFFDh (AY) | ■ | 1 |
| Enable DAC Mode 1 | ■ | ■ | | | | | | | ● | | Ports 0Fh,1Fh, 4Fh, 5Fh (DAC mode 1) | ■ | 1 |
| Enable DAC Mode 2 | ■ | ■ | | | | | | ● | | | Ports F1h, F3h, F9h ,FBh (DAC mode 2) | ■ | 1 |
| Enable Profi/Covox Stereo DAC | ■ | ■ | | | | | ● | | | | Ports 3Fh, 5Fh (DAC stereo-Profi/Covox) | ■ | 1 |
| Enable Covox Stereo DAC | ■ | ■ | | | | ● | | | | | Ports 0Fh, 4Fh (DAC stereo-Covox) | ■ | 1 |
| Enable Pentagon/ATM DAC | ■ | ■ | | | ● | | | | | | Port FBh (DAC mono Pentagon) (SD2 off) | ■ | 1 |
| Enable Covox/GS Mono DAC | ■ | ■ | | ● | | | | | | | Port B3h (DAC mono GS/Covox) | ■ | 1 |
| Enable SPECdrum Mono DAC | ■ | ■ | ● | | | | | | | | Port DFh (DAC mono SPECdrum)[1] | ■ | 1 |

1 Also alias of Kempston port 1Fh

### NextREG 137 (89h) – Expansion Bus Port Decoding Control 4/4 (MSB)

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Enable ULAplus | ■ | ■ | | | | | | | | ● | Ports BF3Bh, FF3Bh (ULAplus) | ■ | 1 |
| Enable DMA | ■ | ■ | | | | | | | ● | | Port 0Bh (Z80DMA) | ■ | 1 |
| Enable Pentagon 1024 Memory | ■ | ■ | | | | | | ● | | | Port EFF7h (Pentagon 1024 memory) | ■ | 0 |
| Enable CTC | ■ | ■ | | | | | ● | | | | Ports 183Bh - 1F3Bh (CTC Channels 0-7) | ■ | 1 |
| Register Reset | ■ | ■ | ● | | | | | | | | Register reset mode (soft or hard reset sel.) | ■ | 1 |

The Internal Port Decoding Enables always apply.
When the Expansion Bus is enabled, the Expansion Bus Port Decoding Enables are logically ANDed with the Internal Enables. A result of 0 for the corresponding bit indicates the internal device is disabled. If the Expansion Bus is enabled, this allows I/O cycles for disabled ports to propagate to the Expansion Bus, otherwise corresponding I/O cycles to the Expansion Bus are filtered.

### NextREG 138 (8Ah) – Expansion Bus I/O Propagate Control

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Port FEh | ■ | ■ | | | | | | | | ● | Propagate port FEh I/O Cycles | ■ | 0 |
| Port 7FFDh | ■ | ■ | | | | | | | ● | | Propagate port 7FFDh I/O Cycles | ■ | 0 |
| Port DFFDh | ■ | ■ | | | | | | ● | | | Propagate port DFFDh I/O Cycles | ■ | 0 |
| Port 1FFDh | ■ | ■ | | | | | ● | | | | Propagate port 1FFDh I/O Cycles | ■ | 0 |
| Port FFh | ■ | ■ | | | | ● | | | | | Propagate port FFh I/O Cycles | ■ | 0 |
| Port EFF7h | ■ | ■ | | | ● | | | | | | Propagate port EFF7h I/O Cycles | ■ | 0 |
| Reserved | ■ | ■ | 0 | 0 | | | | | | | Reserved, must be 0 | ■ | 0 |

If any of the bits are set, I/O cycles for the corresponding ports are propagated to the Expansion Bus when the Expansion Bus is enabled. If the internal port decode is still active, any response sent by devices on the Expansion Bus will be ignored
This allows external peripherals to monitor changes in state inside the ZX Spectrum Next.
Port **FEh** is treated specially, so that external keyboards can be attached. When its propagate bit is set, the value read from the bus will be mixed into keyboard reads on port **FEh**

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | **AFTER SOFT RESET** (Copied into bits D4 through D7) | | |
| ROM 0 (128K) Lock Enable | ■ | ■ | | | | | | | | ● | ROM 0 Lock Enable* | ■ | 0 |
| ROM 1 (48K) Lock Enablen | ■ | ■ | | | | | | | ● | | ROM 1 Lock Enable* | ■ | 0 |
| ALT ROM Availability Switch | ■ | ■ | | | | | | ● | | | ALT ROM visible ONLY during writes | ■ | 0 |
| ALT ROM Enable | ■ | ■ | | | | | ● | | | | ALT ROM Enable | ■ | 0 |
| | | | | | | | | | | | **IMMEDIATE** | | |
| ROM 0 (128K) Lock Enable | ■ | ■ | | | | ● | | | | | ROM 0 Lock Enable* | ■ | 0 |
| ROM 1 (48K) Lock Enable | ■ | ■ | | | | | ● | | | | ROM 1 Lock Enable* | ■ | 0 |
| ALT ROM Availability Switch | ■ | ■ | | | ● | | | | | | ALT ROM visible ONLY during writes | ■ | 0 |
| ALT ROM Enable | ■ | ■ | ● | | | | | | | | ALT ROM Enable | ■ | 0 |

* The locking mechanism applies also if the alternative ROM is not enabled. For the +3/+3e and Next personalities if the two lock bits are **not 0**, then the corresponding ROM page is locked in place. Other personalities (models) use the bits to preferrentially lock the corresponding 48K or 128K ROM in place

## NextREG 142 (8Eh) – ZX Spectrum 128K Memory Mapping

| Group Name | R | W | Data Bits 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| RAM bank 0-15 | ■ | ■ | ● | ● | ● | | | | | | Port 7FFDh D2:D0 | | |
| | | ● | | | | | | | | | Port DFFDh D0 | | |
| RAM bank in ports 7FFDh, DFFDh paging mode | ■ | | | | | | 1 | | | | Returns 1 | | 1 |
| | | ■ | | | | | 0 | | | | No change in RAM bank / MMU 6 / MMU 7 | | |
| | | | | | | | 1 | | | | Change RAM bank | | |
| Normal Paging Mode | ■ | | | | | 0 | | | | | Port 1FFDh D0 | | |
| | | ● | | | | | | | | | Port 7FFDh D4 | | |
| | | ● | | | | | | | | | Port 1FFDh D2 | | |
| allRAM Paging Mode | ■ | | | | | 1 | | | | | Port 1FFDh D0 | | |
| | | ● | | | | | | | | | Port 1FFDh D1 | | |
| | | ● | | | | | | | | | Port 1FFDh D2 | | |

Writes can affect all ports 7FFDh, DFFDh, 1FFDh
Writes always change the ROM / allRAM mapping
Writes immediately change the current MMU mapping as if by Port Write

## NextREG 143 (8Fh) – Memory Mapping Mode Control

| Group Name | R | W | Data Bits 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Memory Mapping Mode* | ■ | ■ | | | | | | | 0 | 0 | Standard ZX 128K / +3[1] | ■ | 0 |
| | | | | | | | | | 0 | 1 | Reserved | | |
| | | | | | | | | | 1 | 0 | Pentagon 512K[2] | | |
| | | | | | | | | | 1 | 1 | Pentagon 1024K[3] | | |
| Reserved | ■ | ■ | 0 | 0 | 0 | 0 | 0 | 0 | | | Reserved, must be 0 | | |

1 Principally ports 7FFDh, DFFDh and 1FFDh
2 Principally port 7FFDh
3 768 Kb on unexpanded Next machines, Principally ports 7FFDh and EFF7h
* The mapping modes affect how ports 7FFDh, DFFDh, 1FFDh and EFF7h carry out memory paging

## NextREG 144 (90h) – PI GPIO Pin Output Enable 1/4

| Group Name | R | W | Data Bits 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Pin 0 | ■ | | | | | | | | | ● | Pin 0 cannot be enabled | | 0 |
| Pin 1 | ■ | | | | | | | | ● | | Pin 1 cannot be enabled | | 0 |
| Pin 2 | ■ | ■ | | | | | | ● | | | Enable | | 0 |
| Pin 3 | ■ | ■ | | | | | ● | | | | Enable | | 0 |
| Pin 4 | ■ | ■ | | | | ● | | | | | Enable | | 0 |
| Pin 5 | ■ | ■ | | | ● | | | | | | Enable | | 0 |
| Pin 6 | ■ | ■ | | ● | | | | | | | Enable | | 0 |
| Pin 7 | ■ | ■ | ● | | | | | | | | Enable | | 0 |

## NextREG 145 (91h) – PI GPIO Pin Output Enable 2/4

| Group Name | R | W | Data Bits 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Pin 8 | ■ | ■ | | | | | | | | ● | Enable | | 0 |
| Pin 9 | ■ | ■ | | | | | | | ● | | Enable | | 0 |
| Pin 10 | ■ | ■ | | | | | | ● | | | Enable | | 0 |
| Pin 11 | ■ | ■ | | | | | ● | | | | Enable | | 0 |
| Pin 12 | ■ | ■ | | | | ● | | | | | Enable | | 0 |
| Pin 13 | ■ | ■ | | | ● | | | | | | Enable | | 0 |
| Pin 14 | ■ | ■ | | ● | | | | | | | Enable | | 0 |
| Pin 15 | ■ | ■ | ● | | | | | | | | Enable | | 0 |

## NextREG 146 (92h) – PI GPIO Pin Output Enable 3/4

| Group Name | R | W | Data Bits 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Pin 16 | ■ | ■ | | | | | | | | ● | Enable | | 0 |
| Pin 17 | ■ | ■ | | | | | | | ● | | Enable | | 0 |
| Pin 18 | ■ | ■ | | | | | | ● | | | Enable | | 0 |
| Pin 19 | ■ | ■ | | | | | ● | | | | Enable | | 0 |
| Pin 20 | ■ | ■ | | | | ● | | | | | Enable | | 0 |
| Pin 21 | ■ | ■ | | | ● | | | | | | Enable | | 0 |
| Pin 22 | ■ | ■ | | ● | | | | | | | Enable | | 0 |
| Pin 23 | ■ | ■ | ● | | | | | | | | Enable | | 0 |

## NextREG 147 (93h) – PI GPIO Pin Output Enable 4/4 (MSB)

| Group Name | R | W | Data Bits 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Pin 24 | ■ | ■ | | | | | | | | ● | Enable | | 0 |
| Pin 25 | ■ | ■ | | | | | | | ● | | Enable | | 0 |
| Pin 26 | ■ | ■ | | | | | | ● | | | Enable | | 0 |
| Pin 27 | ■ | ■ | | | | | ● | | | | Enable | | 0 |

| Group Name | R | W | Data Bits 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Pin 0 | ■ | ■ | | | | | | | | ● | Data | | 1 |
| Pin 1 | ■ | ■ | | | | | | | ● | | Data | | 1 |
| Pin 2 | ■ | ■ | | | | | | ● | | | Data | | 1 |
| Pin 3 | ■ | ■ | | | | | ● | | | | Data | | 1 |
| Pin 4 | ■ | ■ | | | | ● | | | | | Data | | 1 |
| Pin 5 | ■ | ■ | | | ● | | | | | | Data | | 1 |
| Pin 6 | ■ | ■ | | ● | | | | | | | Data | | 1 |
| Pin 7 | ■ | ■ | ● | | | | | | | | Data | | 1 |

## NextREG 153 (99h) – PI GPIO Pin State 2/4

| Group Name | R | W | Data Bits 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Pin 8 | ■ | ■ | | | | | | | | ● | Data | | 1 |
| Pin 9 | ■ | ■ | | | | | | | ● | | Data | | 0 |
| Pin 10 | ■ | ■ | | | | | | ● | | | Data | | 0 |
| Pin 11 | ■ | ■ | | | | | ● | | | | Data | | 0 |
| Pin 12 | ■ | ■ | | | | ● | | | | | Data | | 0 |
| Pin 13 | ■ | ■ | | | ● | | | | | | Data | | 0 |
| Pin 14 | ■ | ■ | | ● | | | | | | | Data | | 0 |
| Pin 15 | ■ | ■ | ● | | | | | | | | Data | | 0 |

## NextREG 154 (9Ah) – PI GPIO Pin State 3/4

| Group Name | R | W | Data Bits 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Pin 16 | ■ | ■ | | | | | | | | ● | Data | | 0 |
| Pin 17 | ■ | ■ | | | | | | | ● | | Data | | 0 |
| Pin 18 | ■ | ■ | | | | | | ● | | | Data | | 0 |
| Pin 19 | ■ | ■ | | | | | ● | | | | Data | | 0 |
| Pin 20 | ■ | ■ | | | | ● | | | | | Data | | 0 |
| Pin 21 | ■ | ■ | | | ● | | | | | | Data | | 0 |
| Pin 22 | ■ | ■ | | ● | | | | | | | Data | | 0 |
| Pin 23 | ■ | ■ | ● | | | | | | | | Data | | 0 |

## NextREG 155 (9Bh) – PI GPIO Pin State 4/4 (MSB)

| Group Name | R | W | Data Bits 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Pin 24 | ■ | ■ | | | | | | | | ● | Data | | 0 |
| Pin 25 | ■ | ■ | | | | | | | ● | | Data | | 0 |
| Pin 26 | ■ | ■ | | | | | | ● | | | Data | | 0 |
| Pin 27 | ■ | ■ | | | | | ● | | | | Data | | 0 |

Writes to the above registers only propagate to the **PI GPIO** when the corresponding pin has its output enabled by **NextREG 144** (**90h**) to **147** (**93h**)

## NextREG 160 (A0h) – PI Peripheral Enable

| Group Name | R | W | Data Bits 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Enable SPI | ■ | ■ | | | | | | | | ● | Enable SPI on GPIO 7,8,9,10,11 * | | 0 |
| Reserved | ■ | ■ | | | | | | | 0 | 0 | Reserved, must be 0 | | |
| Enable I2C | ■ | ■ | | | | | | ● | | | Enable I2C on GPIO 2,3 * | | 0 |
| PI Communication Type | ■ | ■ | | | | | | | 0 | | Connect Rx to GPIO 15, Tx to GPIO 14[1,*] | | 0 |
| | | | | | | | | | 1 | | Connect Rx to GPIO 14, Tx to GPIO 15[2,*] | | |
| Enable UART[3] | ■ | ■ | | | | | | ● | | | Enable UART on GPIO 14,15 * | | 0 |
| Reserved | ■ | ■ | 0 | 0 | | | | | | | Reserved, must be 0 | | |

* Overrides GPIO Enables
1 For communication with Pi HATS
2 For communication with Pi
3 GPIO 16, 17 will act as RTR_n and CTS_n if the UART is in Hardware Flow Control Mode

## NextREG 162 (A2h) – PI I²S Audio Control

| Group Name | R | W | Data Bits 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Redirect to EAR | ■ | ■ | | | | | | | | ● | Direct I²S audio to EAR on port 0xFE | | 0 |
| Reserved | ■ | ■ | | | | | | | | 1 | Reserved, must be 1 | | 1 |
| Mute R control | ■ | ■ | | | | | | | ● | | Mute right side | | 0 |
| Mute L control | ■ | ■ | | | | | | ● | | | Mute left side | | 0 |
| Audio flow direction | ■ | ■ | | | | | 0 | | | | PCM_DOUT to Pi, PCM_DIN from Pi (Hats) | | 0 |
| | | | | | | | 1 | | | | PCM_DOUT from Pi, PCM_DIN to Pi (pi) | | |
| Reserved | ■ | | | | | 0 | | | | | Reserved, must be 0 | | |
| I²S state | ■ | ■ | | | 0 | 0 | | | | | I²S Disabled | | * |
| | | | | | 0 | 1 | | | | | I²S is mono, source R | | |
| | | | | | 1 | 0 | | | | | I²S is mono, source L | | |
| | | | | | 1 | 1 | | | | | I²S is stereo | | |

* Soft reset sets a default of 00

## NextREG 168 (A8h) – ESP WiFi GPIO Output Control

| Group Name | R | W | Data Bits 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| GPIO 0 Output Control | ■ | ■ | | | | | | | | ● | GPIO 0 Output Enable | | 0 |
| GPIO 2 Output Control | ■ | ■ | | | | | | | 0 | | GPIO 2 Output Enable[1] | | 0 |

1 Fixed at 0; GPIO 2 is Read-Only

## NextREG 169 (A9h) – ESP WiFi GPIO Control

| Group Name | R | W | Data Bits 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| GPIO 0 | ■ | ■ | | | | | | | | ● | Read / Write GPIO 0 | | 1 |
| GPIO 2 | ■ | ■ | | | | | | | ● | | Read / Write GPIO 2 | | 1 |

### NextREG 176 (B0h) – Extended Keys 0*

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| RIGHT Status | ■ | | | | | | | | | ● | Key pressed status flag (1 = pressed) | | |
| LEFT Status | ■ | | | | | | | | ● | | Key pressed status flag (1 = pressed) | | |
| DOWN Status | ■ | | | | | | | ● | | | Key pressed status flag (1 = pressed) | | |
| UP Status | ■ | | | | | | ● | | | | Key pressed status flag (1 = pressed) | | |
| . Status | ■ | | | | | ● | | | | | Key pressed status flag (1 = pressed) | | |
| , Status | ■ | | | | ● | | | | | | Key pressed status flag (1 = pressed) | | |
| * Status | ■ | | | ● | | | | | | | Key pressed status flag (1 = pressed) | | |
| ; Status | ■ | | ● | | | | | | | | Key pressed status flag (1 = pressed) | | |

* **NextREG 104** (**68h**) **D4** stops extended keys from making entries into the **8x5** keyboard matrix

### NextREG 177 (B1h) – Extended Keys 1*

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| EXTEND Status | ■ | | | | | | | | | ● | Key pressed status flag (1 = pressed) | | |
| CAPS LOCK Status | ■ | | | | | | | | ● | | Key pressed status flag (1 = pressed) | | |
| GRAPH Status | ■ | | | | | | | ● | | | Key pressed status flag (1 = pressed) | | |
| TRUE VIDEO Status | ■ | | | | | | ● | | | | Key pressed status flag (1 = pressed) | | |
| INV VIDEO Status | ■ | | | | | ● | | | | | Key pressed status flag (1 = pressed) | | |
| BREAK Status | ■ | | | | ● | | | | | | Key pressed status flag (1 = pressed) | | |
| EDIT Status | ■ | | | ● | | | | | | | Key pressed status flag (1 = pressed) | | |
| DELETE Status | ■ | | ● | | | | | | | | Key pressed status flag (1 = pressed) | | |

* **NextREG 104** (**68h**) **D4** stops extended keys from making entries into the **8x5** keyboard matrix

### NextREG 178 (B2h) – Extended MD Pad Buttons

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Left pad MODE Status | ■ | | | | | | | | | ● | Button pressed status flag (1 = pressed) | | |
| Left pad Y Status | ■ | | | | | | | | ● | | Button pressed status flag (1 = pressed) | | |
| Left pad Z Status | ■ | | | | | | | ● | | | Button pressed status flag (1 = pressed) | | |
| Left pad X Status | ■ | | | | | | ● | | | | Button pressed status flag (1 = pressed) | | |
| Right pad MODE Status | ■ | | | | | ● | | | | | Button pressed status flag (1 = pressed) | | |
| Right pad Y Status | ■ | | | | ● | | | | | | Button pressed status flag (1 = pressed) | | |
| Right pad Z Status | ■ | | | ● | | | | | | | Button pressed status flag (1 = pressed) | | |
| Right pad X Status | ■ | | ● | | | | | | | | Button pressed status flag (1 = pressed) | | |

### NextREG 184 (B8) – divMMC Entry Points 0

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0000h Automap Control | ■ | ■ | | | | | | | | ● | 1 to enable automap (instruction fetch) | | 1 |
| 0008h Automap Control | ■ | ■ | | | | | | | ● | | 1 to enable automap (instruction fetch) | | 1 |
| 0010h Automap Control | ■ | ■ | | | | | | ● | | | 1 to enable automap (instruction fetch) | | |
| 0018h Automap Control | ■ | ■ | | | | | ● | | | | 1 to enable automap (instruction fetch) | | |
| 0020h Automap Control | ■ | ■ | | | | ● | | | | | 1 to enable automap (instruction fetch) | | |
| 0028h Automap Control | ■ | ■ | | | ● | | | | | | 1 to enable automap (instruction fetch) | | |
| 0030h Automap Control | ■ | ■ | | ● | | | | | | | 1 to enable automap (instruction fetch) | | |
| 0038h Automap Control | ■ | ■ | ● | | | | | | | | 1 to enable automap (instruction fetch) | | 1 |

### NextREG 185 (B9h) – divMMC Entry Points Valid 0

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Entry point at 0000h | ■ | ■ | | | | | | | | ● | 1 for always* | | 1 |
| Entry point at 0008h | ■ | ■ | | | | | | | ● | | 1 for always* | | 0 |
| Entry point at 0010h | ■ | ■ | | | | | | ● | | | 1 for always* | | 0 |
| Entry point at 0018h | ■ | ■ | | | | | ● | | | | 1 for always* | | 0 |
| Entry point at 0020h | ■ | ■ | | | | ● | | | | | 1 for always* | | 0 |
| Entry point at 0028h | ■ | ■ | | | ● | | | | | | 1 for always* | | 0 |
| Entry point at 0030h | ■ | ■ | | ● | | | | | | | 1 for always* | | 0 |
| Entry point at 0038h | ■ | ■ | ● | | | | | | | | 1 for always* | | 0 |

* Otherwise when ROM3 is present

### NextREG 186 (BAh) – divMMC Entry Points Timing Control 0

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Entry point at 0000h | ■ | ■ | | | | | | | | ● | 1 for instant mapping otherwise delayed | | 0 |
| Entry point at 0008h | ■ | ■ | | | | | | | ● | | 1 for instant mapping otherwise delayed | | 0 |
| Entry point at 0010h | ■ | ■ | | | | | | ● | | | 1 for instant mapping otherwise delayed | | 0 |
| Entry point at 0018h | ■ | ■ | | | | | ● | | | | 1 for instant mapping otherwise delayed | | 0 |
| Entry point at 0020h | ■ | ■ | | | | ● | | | | | 1 for instant mapping otherwise delayed | | 0 |
| Entry point at 0028h | ■ | ■ | | | ● | | | | | | 1 for instant mapping otherwise delayed | | 0 |
| Entry point at 0030h | ■ | ■ | | ● | | | | | | | 1 for instant mapping otherwise delayed | | 0 |
| Entry point at 0038h | ■ | ■ | ● | | | | | | | | 1 for instant mapping otherwise delayed | | 0 |

### NextREG 187 (BBh) – divMMC Entry Points 1

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Entry point at 0066h (delayed) | ■ | ■ | | | | | | | | ● | 1 to enable automap (instr. fetch + button) | | 1 |
| Entry point at 0066h (instant) | ■ | ■ | | | | | | | ● | | 1 to enable automap (instr. fetch + button) | | 0 |
| Entry point at 04C6h (ROM3)[1] | ■ | ■ | | | | | | ● | | | 1 to enable automap (instr. fetch/delayed) | | 1 |
| Entry point at 0562h (ROM3)[1] | ■ | ■ | | | | | ● | | | | 1 to enable automap (instr. fetch/delayed) | | 1 |
| Entry point at 04D7h (ROM3)[2] | ■ | ■ | | | | ● | | | | | 1 to enable automap (instr. fetch/delayed) | | 0 |
| Entry point at 056Ah (ROM3)[2] | ■ | ■ | | | ● | | | | | | 1 to enable automap (instr. fetch/delayed) | | 0 |
| Entry points at 1FF8h:1FFFh | ■ | ■ | | ● | | | | | | | 1 to disable automap (instr. fetch/delayed) | | 1 |
| Entry points at 3Dxxh (ROM3)[3] | ■ | ■ | ● | | | | | | | | 1 to enable automap (instr. fetch/instant) | | 1 |

1 Tape traps. Original divMMC + esxDOS
2 Tape traps. NextZXOS (these offer better compatibility with tape-based software)
3 Used by TR-DOS

## INTERRUPTS

### NextREG 192 (C0h) – Interrupt Control

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Maskable Interrupt Mode | ■ | ■ | | | | | | | | 0 | Pulse | | 0 |
| Current Z80 Interrupt Mode | ■ | | | | | | | | 0 | 0 | Interrupt Mode 0 | | |
| | | | | | | | | | 0 | 1 | Interrupt Mode 1 | | 0 |
| | | | | | | | | | 1 | 0 | Interrupt Mode 2 | | |
| Stackless NMI Enable[1] | ■ | ■ | | | | | | ● | | | Toggle (1=enable, 0=disable) | | 0 |
| Reserved | ■ | ■ | | | | | | 0 | | | Reserved, must be 0 | | 0 |
| IM2 vector | ■ | ■ | ● | ● | ● | ● | | | | | Programmable portion of IM2 vector | | 0 |
| Maskable Interrupt Mode | ■ | ■ | | | | | | | | 1 | Hardware IM2[2] | | 0 |
| Interrupt Vector Generated | ■ | ■ | | | | | 0 | 0 | 0 | 0 | Line interrupt (Highest Priority) | | |
| | | | | | | | 0 | 0 | 0 | 1 | UART 0 Rx | | |
| | | | | | | | 0 | 0 | 1 | 0 | UART 1 Rx | | |
| | | | | | | | 0 | 0 | 1 | 1 | CTC Channel 0 | | |
| | | | | | | | 0 | 1 | 0 | 0 | CTC Channel 1 | | |
| | | | | | | | 0 | 1 | 0 | 1 | CTC Channel 2 | | |
| | | | | | | | 0 | 1 | 1 | 0 | CTC Channel 3 | | |
| | | | | | | | 0 | 1 | 1 | 1 | CTC Channel 4 | | |
| | | | | | | | 1 | 0 | 0 | 0 | CTC Channel 5 | | |
| | | | | | | | 1 | 0 | 0 | 1 | CTC Channel 6 | | |
| | | | | | | | 1 | 0 | 1 | 0 | CTC Channel 7 | | |
| | | | | | | | 1 | 0 | 1 | 1 | ULA | | |
| | | | | | | | 1 | 1 | 0 | 0 | UART 0 Tx | | |
| | | | | | | | 1 | 1 | 0 | 1 | UART 1 Tx (lowest priority) | | |
| | | | ● | ● | ● | | | | | | NextREG 192 (C0h) D7:D5 | | |

1 The return address pushed during an **NMI** acknowledge cycle will be written to **NextREG** instead of memory (the **SP** will be decremented) and the first **RETN** after the acknowledge will take its return address from **NextREG** instead of memory (the **SP** will be incremented) If D3 = 0 and in other circumstances (see advanced hardware manual). **RETN** functions normally
2 In Hardware IM2 Mode the expansion bus is the lowest priority interrupter and if no vector is supplied externally then **255** (**FFh**) will be generated

### NextREG 194 (C2h) – NMI Return Address LSB

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| NMI Return Address | ■ | ■ | ● | ● | ● | ● | ● | ● | ● | ● | 8-bit value (LSB) | | * |

The return address written during an **NMI** acknowledge cycle is always stored in this and the next (**NextREG C3h**) registers

### NextREG 195 (C3h) – NMI Return Address MSB

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| NMI Return Address | ■ | ■ | ● | ● | ● | ● | ● | ● | ● | ● | 8-bit value (MSB) | | * |

The return address written during an **NMI** acknowledge cycle is always stored in this and the previous (**NextREG C2h**) registers

### NextREG 196 (C4h) – Interrupt Enables 0

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ULA | ■ | ■ | | | | | | | | ● | 1 to enable* | | 1 |
| Line | ■ | ■ | | | | | | | ● | | 1 to enable* | | 0 |
| Reserved | ■ | ■ | 0 | 0 | 0 | 0 | 0 | | | | Reserved must be 0 | | 0 |
| Expasion Bus $\overline{INT}$ | ■ | ■ | ● | | | | | | | | 1 to enable | | 1 |

* Aliases of interrupt enable bits in **NextREG 34** (**22h**)
If a device interrupt is disabled, it enters a polled mode

### NextREG 197 (C5h) – Interrupt Enables 1

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| CTC Channel 0 ZC/TO | ■ | ■ | | | | | | | | ● | 1 to enable | | 0 |
| CTC Channel 1 ZC/TO | ■ | ■ | | | | | | | ● | | 1 to enable | | 0 |
| CTC Channel 2 ZC/TO | ■ | ■ | | | | | | ● | | | 1 to enable | | 0 |
| CTC Channel 3 ZC/TO | ■ | ■ | | | | | ● | | | | 1 to enable | | 0 |
| CTC Channel 4 ZC/TO | ■ | ■ | | | | ● | | | | | 1 to enable | | 0 |
| CTC Channel 5 ZC/TO | ■ | ■ | | | ● | | | | | | 1 to enable | | 0 |
| CTC Channel 6 ZC/TO | ■ | ■ | | ● | | | | | | | 1 to enable | | 0 |
| CTC Channel 7 ZC/TO | ■ | ■ | ● | | | | | | | | 1 to enable | | 0 |

If a device interrupt is disabled, it enters a polled mode

### NextREG 198 (C6h) – Interrupt Enables 2

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| UART 0 Rx available[1] | ■ | ■ | | | | | | | | ● | 1 to enable | | 0 |
| UART 0 Rx near-full[1] | ■ | ■ | | | | | | | ● | | 1 to enable | | 0 |
| UART 0 Tx empty | ■ | ■ | | | | | | ● | | | 1 to enable | | 0 |
| Reserved | ■ | ■ | | | | | | 0 | | | Reserved, must be 0 | | 0 |
| UART 1 Rx available[2] | ■ | ■ | | | | | ● | | | | 1 to enable | | 0 |
| UART 1 Rx near-full[2] | ■ | ■ | | | | ● | | | | | 1 to enable | | 0 |
| UART 1 Tx empty | ■ | ■ | | | ● | | | | | | 1 to enable | | 0 |
| Reserved | ■ | ■ | 0 | | | | | | | | Reserved, must be 0 | | 0 |

1 Shared interrupt. **Rx near-full** overrides **Rx available**
2 Shared interrupt. **Rx near-full** overrides **Rx available**
If a device interrupt is disabled, it enters a polled mode

### NextREG 199 (C7h) – Reserved

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Reserved, write 0 | ■ | ■ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Reserved, must be 0 | | 0 |

| R:W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H:D |
|---|---|---|---|---|---|---|---|---|---|---|
| ■ ■ | | | | | | | | ● | Status | |
| ■ ■ | | | | | | | ● | | Status | |
| ■ ■ | 0 | 0 | 0 | 0 | 0 | 0 | | | Reserved must be 0 | |

g. set bits indicate that the device generated an interrupt in the past or an in-
nding
l, set bits clear the status. In Hardware IM2 mode, the status will continue be-
set until the interrupt pending condition is cleared

### D9h) – Interrupt Status 1

Data Bits

| R:W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H:D |
|---|---|---|---|---|---|---|---|---|---|---|
| ) ■ ■ | | | | | | | | ● | Status | |
| ) ■ ■ | | | | | | | ● | | Status | |
| ) ■ ■ | | | | | | ● | | | Status | |
| ) ■ ■ | | | | | ● | | | | Status | |
| ) ■ ■ | | | | ● | | | | | Status | |
| ) ■ ■ | | | ● | | | | | | Status | |
| ) ■ ■ | | ● | | | | | | | Status | |
| ) ■ ■ | ● | | | | | | | | Status | |

g. set bits indicate that the device generated an interrupt in the past or an in-
nding
l, set bits clear the status. In Hardware IM2 mode, the status will continue be-
set until the interrupt pending condition is cleared

### DAh) – Interrupt Status 2

Data Bits

| R:W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H:D |
|---|---|---|---|---|---|---|---|---|---|---|
| ■ ■ | | | | | | | | ● | Status | 0 |
| ■ ■ | | | | | | | ● | | Status | 0 |
| ■ ■ | | | | | | ● | | | Status | 0 |
| ■ ■ | | | | | 0 | | | | Reserved, must be 0 | 0 |
| ■ ■ | | | | ● | | | | | Status | 0 |
| ■ ■ | | | ● | | | | | | Status | 0 |
| ■ ■ | | ● | | | | | | | Status | 0 |
| ■ ■ | 0 | | | | | | | | Reserved, must be 0 | 0 |

upt
upt
g. set bits indicate that the device generated an interrupt in the past or an in-
nding
l, set bits clear the status. In Hardware IM2 mode, the status will continue be-
set until the interrupt pending condition is cleared

### DBh) – Reserved

Data Bits

| R:W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H:D |
|---|---|---|---|---|---|---|---|---|---|---|
| ■ ■ | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | Reserved, must be 1 | |

Fh)

### DCh) – DMA Interrupt Enables 0

Data Bits

| R:W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H:D |
|---|---|---|---|---|---|---|---|---|---|---|
| ■ ■ | | | | | | | | ● | 1 to enable | 0 |
| ■ ■ | | | | | | | ● | | 1 to enable | 0 |
| ■ ■ | | | | | | ● | | | 1 to enable | 0 |

ate the corresponding interrupt will interrupt a DMA operation when in Hard-
ode

### DDh) – DMA Interrupt Enables 1

Data Bits

| R:W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H:D |
|---|---|---|---|---|---|---|---|---|---|---|
| ) ■ ■ | | | | | | | | ● | 1 to enable | 0 |
| ) ■ ■ | | | | | | | ● | | 1 to enable | 0 |
| ) ■ ■ | | | | | | ● | | | 1 to enable | 0 |
| ) ■ ■ | | | | | ● | | | | 1 to enable | 0 |
| ) ■ ■ | | | | ● | | | | | 1 to enable | 0 |
| ) ■ ■ | | | ● | | | | | | 1 to enable | 0 |
| ) ■ ■ | | ● | | | | | | | 1 to enable | 0 |
| ) ■ ■ | ● | | | | | | | | 1 to enable | 0 |

ate the corresponding interrupt will interrupt a DMA operation when in Hard-
ode

### DEh) – DMA Interrupt Enables 2

Data Bits

| R:W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H:D |
|---|---|---|---|---|---|---|---|---|---|---|
| ■ ■ | | | | | | | | ● | 1 to enable | 0 |
| ■ ■ | | | | | | | ● | | 1 to enable | 0 |
| ■ ■ | | | | | | ● | | | 1 to enable | 0 |
| ■ ■ | | | | | 0 | | | | Reserved, must be 0 | 0 |
| ■ ■ | | | | ● | | | | | 1 to enable | 0 |
| ■ ■ | | | ● | | | | | | 1 to enable | 0 |
| ■ ■ | | ● | | | | | | | 1 to enable | 0 |
| ■ ■ | 0 | | | | | | | | Reserved, must be 0 | 0 |

ate the corresponding interrupt will interrupt a DMA operation when in Hard-
ode

| Group Name | R:W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H:D |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Reserved, write 0 | ■ ■ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Reserved, must be 0 | |

* Write 0 (00h)

Since interrupts are only sampled at the end of an instruction by the Z80N, each time the DMA is interrupted, one instrucion of progress is made in the main program

### NextREG 216 (D8h) – I/O Traps (Experimental)

Data Bits

| Group Name | R:W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H:D |
|---|---|---|---|---|---|---|---|---|---|---|---|
| +3 FDC | ■ ■ | | | | | | | | ● | 1 to enable on ports 2FFDh and 3FFDh | 0 |
| Reserved | ■ ■ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | Reserved, must be 0 | |

* An I/O trap generates a Multiface NMI and is indicated in **NextREG 2** (**02h**)
Traps cannot be triggered by the DMA or while the Multiface, divMMC or external NMI is active

### NextREG 217 (D9h) – I/O Traps Write (Experimental)

Data Bits

| Group Name | R:W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H:D |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Data | ■ ■ | ● | ● | ● | ● | ● | ● | ● | ● | 8-bit value | |

Holds data written during trapped I/O write cycle

### NextREG 218 (DAh) – I/O Trap Cause (Experimental)

Data Bits

| Group Name | R:W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H:D |
|---|---|---|---|---|---|---|---|---|---|---|---|
| I/O Trap Cause | ■ ■ | | | | | | | 0 | 0 | None – 0 at the time NR 02 (02h) D4 is 0 | |
| | | | | | | | | 0 | 1 | Port 2FFDh read | |
| | | | | | | | | 1 | 0 | Port 3FFDh read | |
| | | | | | | | | 1 | 1 | Port 3FFDh write | |

* If **NextREG 02** (**02h**):**D4** indicates that an I/O cycle was trapped, this register indicates the cause

### NextREG 240 (F0h) – XDEV (Issue 4 ZX Spectrum Next Only)

Data Bits

| Group Name | R:W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H:D |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Select Mode* | ■ | 1 | | | | | | | | Select Mode Enabled | 1 |
| Currently Selected Device | ■ | | | | | | | 0 | 0 | none | |
| | | | | | | | | 0 | 1 | Xilinx™ DNA (See below) | |
| | | | | | | | | 1 | 0 | Xilinx™ XADC (See below) | |
| Select Device* | ■ | 1 | | | | | | | | Enter Select Mode | |
| | | 0 | | | | | | | | Enter Selected Device Mode | |
| | | 1 | | | | | | | | Change Selected Device | |
| No Device Selected | ■ | | | | | | | 0 | 0 | none | |
| DNA | ■ | | | | | | | | ● | DNA bit (serial stream shifts left)[1] | |
| SELECT / DNA RELOAD | ■ | 1 | | | | | | | | Enter Select Mode or DNA string reload | |
| BUSY | ■ | | 1 | | | | | | | 1 if XADC busy with conversion | |
| EOC | ■ | | | | | | | 1 | | 1 if XADC conv. complete since last read[2] | |
| EOS | ■ | | | | | | | | 1 | 1 if XADC conv. seq. comp. since last read[2] | |
| SELECT | ■ | 1 | | | | | | | | Enter Select Mode | |
| XADC RESET | ■ | | 1 | | | | | | | 1 to reset XADC | |
| XADC CONVST | ■ | | | | | | | | 1 | 1 to start conversion | |

* Re-enter Select Mode at any time by writing to the register with **D7** set
Select a device to communicate with by writing to the register with **D7**:**D6** set then exit Select Mode by clearing **D7**. Thereafter the particular device is attached to the NextREG
1 The first eight bits read will indicate the length of the following DNA bits
2 Read clears

### NextREG 248 (F8h) – XADC Register* (Issue 4 ZX Spectrum Next Only)

Data Bits

| Group Name | R:W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H:D |
|---|---|---|---|---|---|---|---|---|---|---|---|
| DADDR | ■ ■ | ● | ● | ● | ● | ● | ● | ● | ● | XADC DRP register address value | ■ 0 |
| XADC DRP port read/write mode** | ■ ■ | 0 | | | | | | | | Initiate read from XADC DRP port | |
| | | 1 | | | | | | | | Initiate write to XADC DRP port | |

* An XADC Register read or write is initiated by writing to this register
Before initiating a write, store the value in **NextREG 249** (**F9h**) and **NextREG 250** (**FAh**) first
There must be at least **six 28 MHz** cycles after each read or write initiation before the action is completed
** Reads as **0**

### NextREG 249 (F9h) – XADC D0 (Issue 4 ZX Spectrum Next Only)

Data Bits

| Group Name | R:W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H:D |
|---|---|---|---|---|---|---|---|---|---|---|---|
| DRP Data Bus LSB (D7:D0) | ■ ■ | ● | ● | ● | ● | ● | ● | ● | ● | Data | ■ 0 |

* DRP reads store result here and DRP writes take value from here

### NextREG 250 (FAh) – XADC D1 (Issue 4 ZX Spectrum Next Only)

Data Bits

| Group Name | R:W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description | H:D |
|---|---|---|---|---|---|---|---|---|---|---|---|
| DRP Data Bus MSB (D15:D8) | ■ ■ | ● | ● | ● | ● | ● | ● | ● | ● | Data | ■ 0 |

* DRP reads store result here and DRP writes take value from here

### NextREG 255 (FFh) - Reserved (Internal Use Only)

## Other port addresses

As seen in the table at the beginning of this chapter and the discussion about decoding, all even addresses refer to ULA functions. You may find yourself in need to read the keyboard directly from the hardware. As mentioned, part of the ULA's function is to return the state of keypresses. The keyboard is divided in **8** *half-rows* of **5** keys each, each *half-row* having its own port address[5].

| Address Decimal / Hex | Bits | | | | | | | | | | Address Decimal / Hex |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | D0 | D1 | D2 | D3 | D4 | D4 | D3 | D2 | D1 | D0 | |
| 63486 / F7FEh | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0 | 61438 / EFFEh |
| 64510 / FBFEh | Q | W | E | R | T | Y | U | I | O | P | 57342 / DFFEh |
| 65022 / FDFEh | A | S | D | F | G | H | J | K | L | Enter | 49150 / BFFEh |
| 65278 / FEFEh | Caps | Z | X | C | V | B | N | M | Sym | Space | 32766 / 7FFEh |

The diagram above neatly illustrates how the keyboard matrix is separated into *half-rows* (demarcated by the thick line in the middle). Pay attention to how bits are mirrored going from the outside of the keyboard to the inside.

The address of each *half-row* in the diagram is calculated as: **254 + 256*(255 - 2$^{n}$)**.

In the formula above, **n** is the number of *half-row* which starts with **0** at the bottom right and moves in a counterclockwise manner with each successive *half-row* increasing by **1**.

In the byte read in, bits **D0** to **D4** stand for each of the five keys in the given *half-row* – **D0** for the outside key and **D4** for the one nearest the middle. The bit is **0** if the key is pressed and **1** if it is not.

For example to find the value of the **CAPS SHIFT** key, you can do:

```
PRINT %IN 65278 & @1
```

Writing a value using **OUT** to the ULA (Port **254** / **FEh**) controls other hardware as well. You can drive the beeper with **D4**, the MIC socket with **D3**, read the **EAR** socket with **D6** and modify the **BORDER** colour using bits **D0**,**D1** and **D2**. For example to make the border a nice magenta colour you can:

```
OUT 254, %@00000011
```

Port addresses **32765** (**7FFDh**), **8189** (**1FFDh**) and **57341** (**DFFDh**) control the extra memory. Executing an **OUT** to these ports from *NextBASIC* without knowing the ramifications will nearly always cause the computer to crash, losing any program and data. These ports are write-only, i.e. you cannot determine the current state of the paging by an **IN** instruction. This is why the BANKM system variable is always kept up to date with the last value output to this port. Check the last section in this chapter as well as *Chapter 23 – The Memory* where we examine the banking system in detail.

Writing to port **65533** (**FFFDh**) will select a particular PSG register (on the AY sound chip) and writing to port **49149** (**BFFDh**) will send a particular value to that register. Reading from port **65533** (**FFFDh**) returns the value stored in the selected register. Judicious use of these two registers can allow sounds to be generated while *NextBASIC* gets on with something else.

The section that follows describes all ZX Spectrum Next – specific hardware ports; addressing them is via **OUT** and **IN** commands.

5 *Extended keys are combinations of tther keys, so they need to be read as those key combinations. For example EXTEND is CAPS SHIFT + SYMBOL SHIFT.*

# The ZX Spectrum Next Hardware Ports List

## NOTE

The following Hardware Ports are not listed:
ULA 254 (FEh), Legacy Memory Paging 3765 (7FFDh), 8189 (1FFDh),
57341 (DFFDh), 61431 (EFF7h), Layer 2 4667 (123Bh)
and ULAplus 65339 (FF3Bh) / 48955 (BF3B)
Ports are arranged according to their function.
Please also consult the **ports.txt** document found in your **System/Next™** distribution for a complete list of ports

## Input / Output / Legacy Video

### Port 255 (FFh) – Timex SCLD ULA Extensions*

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Screen Mode Select | ■ | ■ | | | | | | | 0 | 0 | 0 | Std: DFILE0+COLOURFILE0 @ 4000h |
| | | | | | | | | | 0 | 0 | 1 | Shd: DFILE1+COLOURFILE1 @ 6000h |
| | | | | | | | | | 0 | 1 | 0 | HC: DFILE @ 4000h, COLOURFILE @ 6000h |
| | | | | | | | | | 1 | 1 | 0 | HR: even DFILE @ 4000h, odd DFILE @6000h |
| HiRes Colour Scheme Select | ■ | ■ | | | | 0 | 0 | 0 | | | | BRIGHT Black on White |
| | | | | | | 0 | 0 | 1 | | | | BRIGHT Blue on Yellow |
| | | | | | | 0 | 1 | 0 | | | | BRIGHT Red on Cyan |
| | | | | | | 0 | 1 | 1 | | | | BRIGHT Magenta on Green |
| | | | | | | 1 | 0 | 0 | | | | BRIGHT Green on Magenta |
| | | | | | | 1 | 0 | 1 | | | | BRIGHT Cyan on Red |
| | | | | | | 1 | 1 | 0 | | | | BRIGHT Yellow on Blue |
| | | | | | | 1 | 1 | 1 | | | | BRIGHT White on Black |
| Frame Interrrupt Control | ■ | ■ | ● | | | | | | | | | 1 to Disable ULA Frame Interrupt |
| Timex MMU Select[1] | ■ | ■ | ● | | | | | | | | | Timex Horizontal MMU Bank Select |

* Only readable if **NR 8 (08h):D2 = 1**
[1] Not implemented on the ZX Spectrum Next

### Port 64479 (FBDFh) – Kempston Mouse X position

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Current X Position | ■ | | ● | ● | ● | ● | ● | ● | ● | ● | Value 0 – 255* |

* Returns the current X position of the mouse **0 - 255**.
The value wraps from **255** to **0** on a right movement and from **0** to **255** on a left movement.

### Port 65503 (FFDFh) – Kempston Mouse Y position

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Current Y Position | ■ | | ● | ● | ● | ● | ● | ● | ● | ● | Value 0 - 255* |

* Returns the current Y position of the mouse **0 - 255**.
The value decrements and wraps from **255** to **0** on a downward movement and increments and wraps from **0** to **255** on an upward movement.

### Port 64223 (FADFh) – Kempston Mouse Button Status

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Mouse Button Flags[1] | ■ | | | | | | | | | ● | Left Mouse button status |
| | | | | | | | | | ● | | Right Mouse button status |
| | | | | | | | | ● | | | Middle Mouse button status |
| Mouse Wheel Position[2] | ■ | | ● | ● | ● | ● | | | | | Mouse Wheel position (Wraps) |

[1] Pressed = **1**, Not Pressed = **0**
[2] Value **0** to **15**, Upwards scroll **15** to **0** – wraps to **15**; Downards scroll **0** to **15** – wraps to **0**

### Port 31 (1Fh) – Kempston Joystick 1 / Mega Drive Pad 1 Status

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Joystick Movement Status | ■ | | | | | | | | | ● | Right (pin 4) |
| | | | | | | | | | ● | | Left (pin 3) |
| | | | | | | | | ● | | | Down (pin 2) |
| | | | | | | | ● | | | | Up (pin 1) |
| Joystick Button Status | ■ | | | | | ● | | | | | Fire 1 (MD = B) (pin 6) |
| | | | | | ● | | | | | | Fire 2 (MD = C) (pin 9) |
| | | | | ● | | | | | | | MD = A (0 on Kempston) |
| | | | ● | | | | | | | | MD Start (0 on Kempston) |

Joystick ports can be placed in I/O mode via NR 11 (0Bh)
Kempston joysticks and Mega Drive Pads share ports but MD pads use more bits
The X/Y/Z buttons on MD pads can be read via NR 178 (B2h)
All twelve MD pad buttons can be assigned to the keyboard – See NR 05 (05h)
When using an MD pad in more limited joystick modes, the excess buttons can generate keypresses if so programmed

### Port 55 (37h) – Kempston Joystick 2 / Mega Drive Pad 2 Status

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Joystick Movement Status | ■ | | | | | | | | | ● | Right (pin 4) |
| | | | | | | | | | ● | | Left (pin 3) |
| | | | | | | | | ● | | | Down (pin 2) |
| | | | | | | | ● | | | | Up (pin 1) |
| Joystick Button Status | ■ | | | | | ● | | | | | Fire 1 (MD = B) (pin 6) |
| | | | | | ● | | | | | | Fire 2 (MD = C) (pin 9) |
| | | | | ● | | | | | | | MD = A (0 on Kempston) |
| | | | ● | | | | | | | | MD Start (0 on Kempston) |

Joystick ports can be placed in I/O mode via NR 11 (0Bh)
Kempston joysticks and Mega Drive Pads share ports but MD pads use more bits
The X/Y/Z buttons on MD pads can be read via NR 178 (B2h)
All twelve MD pad buttons can be assigned to the keyboard – See NR 05 (05h)
When using an MD pad in more limited joystick modes, the excess buttons can generate keypresses if so programmed

## Audio

### Port 65533 (FFFDh) – PSG Control and Register Select

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Selected Register Status | ■ | | ● | ● | ● | ● | ● | ● | ● | ● | Value in selected register of active PSG |
| Active PSG Control | | | | | | | | | 0 | 0 | Reserved |
| | ■ | | | | | | | | 1 | 1 | PSG 0 made active* |
| | | | | | | | | | 1 | 0 | PSG 1 made active |
| | | | | | | | | | 0 | 1 | PSG 2 made active |
| Stereo Channel Control[1] | | | | | | | | 1 | | | Reserved, must be 1 |
| | | | | | | | 1 | | | | Reserved, must be 1 |
| | | | | | | 1 | | | | | Reserved, must be 1 |
| | ■ | | | | ● | | | | | | Right Channel Enable |
| | | | | ● | | | | | | | Left Channel Enable |
| | | | 1 | | | | | | | | Reserved, must be 1 |
| Register Select[2] | | | ● | ● | ● | ● | | | | | If 0000 selects a register from the Active PSG |

* Default value
[1] If **NR 8 (08h):D1** = 1 and **D7** through **D4** are not **0000**

### Port 49149 (BFFDh) – PSG Data

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Selected Register Status[1] | ■ | | ● | ● | ● | ● | ● | ● | ● | ● | Value in selected register of active PSG |
| Active PSG Register Data | ■ | ■ | ● | ● | ● | ● | ● | ● | ● | ● | Value to write to the register |

[1] Readable if machine type is ZX Spectrum Next or ZX Spectrum +3 only.

### Port 49141 (BFF5h) – PSG Info

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Currently selected PSG register | ■ | | | | | ● | ● | ● | ● | ● | Value of selected register of active PSG |
| Reserved | ■ | | | | ● | | | | | | Reserved |
| Active PSG | ■ | | 1 | 1 | | | | | | | PSG 0 is active |
| | | | 1 | 0 | | | | | | | PSG 1 is active |
| | | | 0 | 1 | | | | | | | PSG 2 is active |

### Ports 251[1], 223[2], 31[3], 241[4], 63[5] (FBh, DFh, 1Fh, F1h, 3Fh) – DAC Channel A (Left)

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
|---|---|---|---|---|---|---|---|---|---|---|---|
| DAC output | | ■ | ● | ● | ● | ● | ● | ● | ● | ● | 8-bit unsigned sample value |

All DACs originate from various ZX Spectrum peripherals and compatible models and are kept for compatibility. DACs are enabled by setting **NR 8 (08h):D3** = 1
[1] Found in Pentagon and ATM
[2] Found in SpecDRUM™
[3] Found in SoundDrive 1
[4] Found in SoundDrive 2
[5] Found in Profi Covox

### Ports 179[1], 15[2], 243[3] (B3h, 0Fh, F3h) – DAC Channel B (Left)

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
|---|---|---|---|---|---|---|---|---|---|---|---|
| DAC output | | ■ | ● | ● | ● | ● | ● | ● | ● | ● | 8-bit unsigned sample value |

All DACs originate from various ZX Spectrum peripherals and compatible models and are kept for compatibility. DACs are enabled by setting **NR 8 (08h):D3** = 1
[1] Found in GS Govox
[2] Found in SoundDrive 1 and Covox
[3] Found in SoundDrive 2

### Ports 179[1], 79[2], 249[3] (B3h, 4Fh, F9h) – DAC Channel C (Right)

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
|---|---|---|---|---|---|---|---|---|---|---|---|
| DAC output | | ■ | ● | ● | ● | ● | ● | ● | ● | ● | 8-bit unsigned sample value |

All DACs originate from various ZX Spectrum peripherals and compatible models and are kept for compatibility. DACs are enabled by setting **NR 8 (08h):D3** = 1
[1] Found in GS Govox
[2] Found in SoundDrive 1 and Covox
[3] Found in SoundDrive 2

### Ports 251[1], 223[2], 95[3] (FBh, DFh, 5Fh) – DAC Channel D (Right)

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
|---|---|---|---|---|---|---|---|---|---|---|---|
| DAC output | | ■ | ● | ● | ● | ● | ● | ● | ● | ● | 8-bit unsigned sample value |

All DACs originate from various ZX Spectrum peripherals and compatible models and are kept for compatibility. DACs are enabled by setting **NR 8 (08h):D3** = 1
[1] Found in Pentagon, ATM and SoundDrive 2
[2] Found in SpecDRUM™
[3] Found in SoundDrive 1 and Profi Covox

## Storage

### Port 227 (E3h) – divMMC Control

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
|---|---|---|---|---|---|---|---|---|---|---|---|
| RAM bank Control | ■ | ■ | | | | | ● | ● | ● | ● | Memory Bank Select for 8K - 16K region |
| MapRAM Control[1] | ■ | ■ | | | | ● | | | | | MapRAM Enable |
| ConMEM Control[2] | ■ | ■ | ● | | | | | | | | ConMEM Enable |

[1] Can only be set once. On the original divMMC only a power cycle can reset it. The ZX Spectrum Next implementation uses **NR 9** (**09h**) where **D3** can be set,in order to reset this bit. When set, it replaces the expected esxDOS ROM with divMMC RAM bank **3**
[2] Can be used to manually control divMMC mapping. When set it maps in divMMC; **0K - 8K** will contain the esxDOS ROM. **8K - 16K** will contain the selected divMMC bank (from **D0** through **D3**). The divMMC automatically maps itself in when instruction fetches hit specific addresses in the ROM. When this happens, the esxDOS ROM (or divMMC bank **3** if mapRAM is set) appears in **0K - 8K** and the selected divMMC bank appears as RAM in **8K - 16K**. DivMMC automapping is normally disabled by NextZXOS. See **NR 6** (**06h**):**D4**. The divMMC has been enhanced, in order to add entry points and make them programmable.

## Communication

### Port 4155 (103Bh) – I²C SCL

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
|---|---|---|---|---|---|---|---|---|---|---|---|
| I²C Clock Line Control | ■ | ■ | | | | | | | | ● | State of the Clock Line |

### Port 4411 (113Bh) – I²C SDA

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
|---|---|---|---|---|---|---|---|---|---|---|---|
| I²C Data Line Control | ■ | ■ | | | | | | | | ● | State of the Data Line |

### Port 231 (E7h) – SPI CS*

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
|---|---|---|---|---|---|---|---|---|---|---|---|
| SD Card 0 Select | ■ | ■ | | | | | | | | ● | Select SD Card 0 |
| SD Card 1 Select | ■ | ■ | | | | | | | ● | | Select SD Card 1 |
| Pi SPI 0 Select[1] | ■ | ■ | | | | | | ● | | | Select Pi SPI 0 on the GPIO pins |
| Pi SPI 1 Select[1] | ■ | ■ | | | | | ● | | | | Select Pi SPI 1 on the GPIO pins |
| FPGA Flash Select | ■ | ■ | ● | | | | | | | | Select the FPGA Flash ROM (Internal Use Only) |

* The SPI port's data lines are active low (0 to select).
[1] Pi GPIO must be configured for SPI. See **NR 160** (**A0h**)
Five devices are connected to the SPI interface. The ZX Spectrum Next must be SPI master.
Only one of **D0** through **D3** can be **0** at one time. If not, the result will be no device selected

### Port 235 (EBh) – SPI Data

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
|---|---|---|---|---|---|---|---|---|---|---|---|
| SPI Data | ■ | ■ | ● | ● | ● | ● | ● | ● | ● | ● | Read/Write data to the selected SPI device |

### Port 5435 (153Bh) – UART Control

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Prescalar MSB Value | ■ | ■ | | | | | | ● | ● | ● | Baud rate prescalar MSB |
| Prescalar MSB Write Enable | ■ | ■ | | | | ● | | | | | D2:D0 write enable |
| UART Select** | ■ | ■ | | 0 | | | | | | | ESP UART Select |
| | | | | 1 | | | | | | | PI0 UART Select* |

* Pi GPIO must be configured for UART. See **NR 160** (**A0h**)
** Either UART can be redirected to the joystick ports, see **NR 11** (**0Bh**)

### Port 4923 (133Bh) – UART Transmit

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Read buffer status flag | ■ | | | | | | | | | ● | Set if read buffer contains received bytes |
| Transmitter busy flag | ■ | | | | | | | | ● | | Set if the transmit buffer is full |
| Read buffer full flag | ■ | | | | | | | ● | | | Set if the read buffer overflowed* |
| Read buffer near-full flag | ■ | | | | | | ● | | | | Set if the read buffer in near-full (¾) |
| Transmit buffer empty flag | ■ | | | | | ● | | | | | Set if the transmit buffer is empty |
| Receive data after error flag | ■ | | | | ● | | | | | | Set if the next Rx byte was recv'd after an error** |
| Receive framing error flag | ■ | | | ● | | | | | | | Set if the Rx experienced a framing error*** |
| Receive in break flag | ■ | | ● | | | | | | | | Set if the Rx is in a break condition**** |
| Data Transmit | ■ | ● | ● | ● | ● | ● | ● | ● | ● | ● | Send a byte to the connected device |

* Clears on read
** Framing or overflow
*** Clears on read; includes parity and stop bit errors
**** External device has held Tx=0 for at least 20 bit periods
Both UARTs have a 64-byte transmit buffer

### Port 5179 (143Bh) – UART Receive

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Data Receive | ■ | | ● | ● | ● | ● | ● | ● | ● | ● | Reads a byte from the receive buffer* |
| Prescalar LSB Value[1] | ■ | | 0 | ● | ● | ● | ● | ● | ● | ● | Lower 7 bits of the 14-bit prescalar value LSB |
| | | | 1 | ● | ● | ● | ● | ● | ● | ● | Upper 7 bits of the 14-bit prescalar value LSB |

* If the buffer is empty **0** is returned.
[1] The UART's baud rate is determined by the prescalar according to this formula:
Prescalar = $F_{sys}$ / **baudrate**; $F_{sys}$ = System Clock from **NR 17** (**11h**),
Example: If the system is on a Digital display, **NR 17** (**11h**) indicates that $F_{sys}$ = 27000000.
The prescalar for a baud rate of **115200** is **27000000** / **115200** = **234**
Both UARTs have a 512-byte receive buffer

### Port 5691 (163Bh) – UART Frame

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Stop bits control | ■ | ■ | | | | | | | | ● | Set if 2 stop bits, clear for 1 stop bit |
| Parity type control | ■ | ■ | | | | | | | ● | | Set if odd parity, clear for even parity |
| Parity check control | ■ | ■ | | | | | | ● | | | Set to enable parity check |
| Frame size (bits) control | ■ | ■ | | | | 0 | 0 | | | | 5 bits |
| | | | | | | 0 | 1 | | | | 6 bits |
| | | | | | | 1 | 0 | | | | 7 bits |
| | | | | | | 1 | 1 | | | | 8 bits |
| Hardware Flow control | ■ | ■ | | | ● | | | | | | Set enables hardware flow control* |
| Transmit break control** | ■ | ■ | | ● | | | | | | | Send a byte to the connected device |
| Reset Rx and Tx Control | ■ | ■ | ● | | | | | | | | Set to immediately reset Rx, Tx and empty FIFOs |

* Honoured only on an Issue 4 ESP module, no effect on Issue 2 boards. In joystick I/O mode only CTS is available
** Asserts break on Tx when Tx becomes idle

## Sprites

### Port 12347 (303Bh) – Sprite Slot Select*[1]

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Sprite Collision flag | ■ | | | | | | | | | ● | Set if any two displayed sprites collide on screen |
| Max. No. of Sprites per line flag | ■ | | | | | | | | | ● | Set if maximum no. of sprites per line exceeded |
| Current Pattern Index Select[2] | ■ | ● | | ● | ● | ● | ● | ● | ● | ● | Sets Current Pattern Index |
| Current Sprite Index Select[1] | ■ | | ● | ● | ● | ● | ● | ● | ● | ● | Sets Current Sprite (0 - 127) |

* Reading the port clears all flags
[1] The current sprite and pattern index are separate quantities internally
[2] The pattern index is 6-bit in bits **D0** through **D5** and selects pattern **0** - **63** in the pattern RAM. Each pattern is **256** bytes long. **D7** can be used to offset **128** bytes halfway through the pattern; this accommodates 4-bit sprites whose patterns are **128** bytes in size.

### Port (57h) – Sprite Attributes

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Attribute Data | ■ | ● | ● | ● | ● | ● | ● | ● | ● | ● | Attribute Data |

Writes the current sprite's attributes. Each sprite has either 4 or 5 attributes and after all are written, the current sprite pointer is advanced to the next sprite. The pointer wraps from **127** to **0**.

### Port (5Bh) – Sprite Pattern

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Pattern Data | ■ | ● | ● | ● | ● | ● | ● | ● | ● | ● | Pattern Data |

Writes a byte to the current pattern address and advances the current address by one. The pattern address is changed by writing the pattern index in port **12347** (**303Bh**). A pattern index indicates the start of a 256-byte range of data used to define an 8-bit sprite pattern or a 128-byte range of data use to define a 4-bit sprite pattern.

## DMA

### Port 107 (6Bh) / 11 (0Bh) – zxnDMA / Z80DMA

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
|---|---|---|---|---|---|---|---|---|---|---|---|
| DMA Control | ■ | ■ | ● | ● | ● | ● | ● | ● | ● | ● | DMA command value |

The zxnDMA implements a subset of the Zilog Z80DMA architecture while adding a burst mode primarily used to play digital music. Accessing the DMA via port **11** (**0Bh**) rather than **107** (**6Bh**) selects the DMA compatibility mode (Z80DMA). **See https://www.specnext.com/the-zxndma/**

## Layer 2 Graphics

### Port 4667 (123Bh) – Layer 2 Control

| Group Name | R | W | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Description |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Memory Write Mapping Control | ■ | ■ | | | | | | | | ● | Enable Mapping for Memory Writes |
| Layer 2 Display Control | ■ | ■ | | | | | | | ● | | Enable Layer 2 Display |
| Memory Read Mapping Control | ■ | ■ | | | | | | ● | | | Enable Mapping for Memory Reads |
| Active/Shadow Control | ■ | ■ | | | | | | 0 | | | Map Active Layer 2[1] |
| | | | | | | | | 1 | | | Map Shadow Layer 2[2] |
| Reserved | | ■ | | | 0 | 0 | | | | | Reserved, Must be 0 |
| Layer 2 Map Type Select | ■ | ■ | 0 | 0 | | | | | | | First 16K of Layer 2 in the bottom 16K |
| | | | 0 | 1 | | | | | | | Second 16K of Layer 2 in the bottom 16K |
| | | | 1 | 0 | | | | | | | Third 16K of Layer 2 in the bottom 16K |
| | | | 1 | 1 | | | | | | | First 48K of Layer 2 in the bottom 48K |

* Memory pointed at by **NR 18** (**12h**) or **NR 19** (**13 h**) can be mapped into the lower 16K or 48K if Layer 2 memory mapping is enabled in **D2** and/or **D0**. This mechanism is separate from MMU and will overlay the paging state set by MMU but only if the memory access type matches the enable condition (Read-only, Write-only)
[1] See **NR 18** (**12h**)
[2] See **NR 19** (**13h**)

## Memory Mapping Modes

The current memory mapping is maintained by the hardware in eight MMUs with each MMU, numbered MMU0 – MMU7, holding an 8K page number currently mapped into corresponding eight 8K slots of the Z80's 64K memory space. The individual MMUs can be programmed via **NextREGs 80 – 87** (**50h – 57h**).

The traditional banking mechanism used in various Spectrum models is via ports 7FFDh, DFFDh, 1FFDh and EFF7h with writes to these ports making changes to one or more MMUs in the hardware.

The 128K Spectrum used port 7FFDh, the +3 used 7FFDh and 1FFDh and the ZX Spectrum Next extends that with port DFFDh to increase the range of 16K banks that can be paged into the top 16K.

Russian ZX Spectrum models extended the banking in various ways to support up to 1MB of memory. The ZX Spectrum Next, also supports some of these banking methods which modify the behaviour of ports **7FFDh** and **DFFDh**. The current memory mapping mode is held in **NextREG 143** (**8Fh**) and can be one of standard ZX Spectrum, Pentagon 512K or Pentagon 1024K. The ZX Spectrum Next normally operates in standard Spectrum mode.

In standard spectrum mode, ports **7FFDh** and **1FFDh** operate as on the +3 and as described above.

Port **DFFDh** adds four bits to extend the three bit bank number in bits **D2:D0** of port **7FFDh** so that the full range of the ZX Spectrum

Next's memory can be paged into the top 16K.

Also in this mode, **D3** of port **EFF7h** can be set to place 16K bank 0 into the bottom 16K, overlaying the ROM.

Writes to ports **7FFDh**, **DFFDh**, **1FFDh** and **EFF7h**, will cause the bottom 16K and the top 16K to remap; if allRAM mode is selected, the entire 64K remaps.

In Pentagon 512K mode, port **7FFDh** is extended so that bits **D7:D6** augment the bank number in bits **D2:D0** to form a 5-bit bank number for the top 16K.

On the ZX Spectrum Next, port **1FFDh** continues to function (allRAM mode and ROM selection) and **D3** of port **EFF7h** can be set to map 16K bank 0 on top of ROM in the bottom 16K.

Writes to ports **7FFDh**, **DFFDh**, **1FFDh** and **EFF7h** will cause the bottom 16K and the top 16K to remap; if allRAM mode is selected, the entire 64K remaps.

In Pentagon 1024K mode, port **EFF7h D2 = 0** enables Pentagon 1024K mapping and port **EFF7h D2 = 1** selects standard ZX Spectrum mapping as described above.

With Pentagon 1024K mode active, port **7FFDh** is unlocked and **D5** (the lock bit) is repurposed as another bank bit.

The bank number paged into the top 16K is taken as port **7FFDh** bits {**D5**,**D7**,**D6**,**D2**,**D1**,**D0**} to reach 1MB in 16K banks.

On the Next, port **1FFDh** continues to function (allRAM mode and ROM selection) and **D3** of port **EFF7h** can be set to map 16K bank 0 on top of ROM in the bottom 16K.

Writes to ports **7FFDh**, **DFFDh**, **1FFDh** and **EFF7h** will cause the bottom 16K and the top 16K to remap; if allRAM mode is selected, the entire 64k remaps.

For 100% compatibility with the original banking methods listed above, extra ports like **1FFDh** can be disabled using the internal port decodes with **NextREGs 130 - 133** (**82h - 85h**).



## The Expansion Bus

The Expansion Bus is found in the back of the ZX Spectrum Next and exposes its CPU to the world. As it too gets addressed by **IN** and **OUT** commands, it is listed below:

| Bottom Side | | | | Top Side |
|---|---|---|---|---|
| A11 | 28 | 28 | Reserved | |
| A9 | 27 | 27 | A10 | |
| $\overline{\text{BUSACK}}$ | 26 | 26 | A8 | |
| $\overline{\text{ROMCS}}$ | 25 | 25 | $\overline{\text{RFSH}}$ | |
| A4 | 24 | 24 | $\overline{\text{M1}}$ | |
| A5 | 23 | 23 | NC | |
| A6 | 22 | 22 | NC | |
| A7 | 21 | 21 | $\overline{\text{WAIT}}$ | |
| $\overline{\text{RESET}}$ | 20 | 20 | NC | |
| BUSREQ[6] | 19 | 19 | $\overline{\text{WR}}$ | |
| NC | 18 | 18 | $\overline{\text{RD}}$ | |
| NC | 17 | 17 | $\overline{\text{IORQ}}$ | |
| Reserved | 16 | 16 | $\overline{\text{MREQ}}$ | |
| $\overline{\text{ROMCS}}$ | 15 | 15 | $\overline{\text{HALT}}$ | |
| GND | 14 | 14 | $\overline{\text{NMI}}$ | |
| $\overline{\text{IORQULA}}$ | 13 | 13 | $\overline{\text{INT}}$ | |
| A3 | 12 | 12 | D4 | |
| A2 | 11 | 11 | D3 | |
| A1 | 10 | 10 | D5 | |
| A0 | 9 | 9 | D6 | |
| $\overline{\text{CLK}}$ | 8 | 8 | D2 | |
| GND | 7 | 7 | D1 | |
| GND | 6 | 6 | D0 | |
| **Key** | 5 | 5 | Key | |
| +9V (PSU)[7] | 4 | 4 | $\overline{\text{ROMCS}}$ | |
| +5V | 3 | 3 | D7 | |
| A12 | 2 | 2 | A13 | |
| A14 | 1 | 1 | A15 | |

*The ZX Spectrum Next Expansion Bus*

6 *BUSREQ is incorrectly active High oin Issue 2 but fixed on Issue 4*

7 *This pin receives the unregulated power from the PSU line. If you plug a higher voltage PSU, that voltage will be present at that pin and may damage your peripherals*

# Chapter 23 – The Memory

## Overview

In previous chapters, we talked about binary code, bytes, words and long words. We also discussed strings, floating point and integer numbers. It's time to go into more detail and explore how your computer stores information we put into it.

The kind of data we're processing makes absolutely no difference to the computer. Whether it's music, a game or a document, it ends up as a series of ones and zeros organised as bytes and stored in memory. We can rely on *NextBASIC* to manage that information or we can do it ourselves as long as we know how!

The ZX Spectrum Next is an 8-bit computer with a 16-bit Address Bus. That means that it stores and manipulates information in 8-bit bytes, and can see at most 65536 of these bytes at one time. Hold onto this information for now as it's important.

## ROM and RAM

Memory can be categorized into two kinds: ROM and RAM. ROM (read-only memory) cannot be written to whereas RAM (random access memory) can be both read and written. RAM is where things like the program and display contents are stored because they can change while the computer is running. ROM can be used to hold something permanent like the *NextBasic* interpreter or *NextZXOS*. You may have picked up on the discussion of the ROM earlier and may have been wondering how we can load a ROM from a file as described in various places around this book, when ROM is supposed to be permanent and read-only (see for example *Chapter 1*).

The truth of the matter is that, although the ZX Spectrum Next contains a physical ROM chip, this has nothing to do with the ZX Spectrum Next's operation. The physical ROM is used to configure the Xilinx FPGA and a small amount is used to store a program that configures the machine on boot. The ZX Spectrum Next itself only sees RAM memory supplied by up to four 512K SRAM chips. The unexpanded model has two chips present for a total of 1024K of memory and the expanded model has four for a total of 2048K memory (See *Chapter 22* on how to upgrade the RAM to the maximum possible). The ROM contents are loaded into a portion of this RAM and then the hardware is instructed to make that portion read-only. So after the machine boots, those areas of RAM behave just like ROM because running programs cannot change anything stored there. This reproduces the behaviour of the original Spectrums which did use physical ROMs to store the basic interpreter. In other words, for the purposes of *NextBASIC* and *NextZXOS* the ZX Spectrum Next indeed has ROM.

## The Memory Map

In the introduction of this chapter we talked about how the ZX Spectrum Next has a 16-bit Address Bus and how this fact means the computer can see **65536** bytes (64 Kilobytes) of memory, a figure that includes both ROM and RAM. That is enough to generate the obvious question: But my computer has 16 (or 32) times as much memory, what's the point of having it? And you would be absolutely right to ask this!

The answer to that question is that the computer uses a memory access technique known as bank switching. In this technique there's a distinction between the maximum addressable memory (the amount of memory that the CPU can see, ie 64K in our case) and the amount of physical memory in the system. In the ZX Spectrum Next's case, the physical memory is divided into equally sized portions called banks and the 64K of memory that the computer can see is also divided into the same sized portions called slots. A virtual map of sorts is constructed that tells the hardware what physical memory bank appears in each of the 64K's slots. We shall refer to this virtual map as *the memory map*. Whenever information located in physical memory is required, the specific physical bank that holds it is entered into the memory map in one of its slots so that the CPU can see the bank in the slot's address range. Paging in the new bank replaces whatever was there before be-

cause the CPU is given a new window in to a different bank in physical memory. This way the usable physical memory can far exceed the memory the CPU can normally see while, at the same time, older software is completely unaware and will continue to run properly without performing any bank switching.

## Memory Management

There are two banking schemes employed in the ZX Spectrum Next: Standard and MMU-based banking. The Standard scheme is inherited from the +3 and the other 128K Spectrum models. The MMU scheme co-exists with the Standard scheme but it is unique to the ZX Spectrum Next.

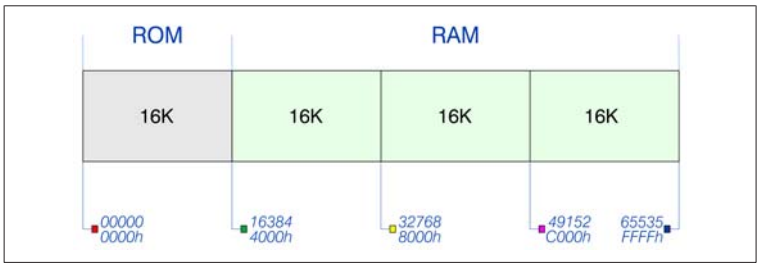



*Fig. 46 – Standard (NextBASIC) memory map*

As you can see, in the memory map *NextBASIC* uses, the available 64K of addressable memory is divided into four slots of 16K each with the bottom slot always occupied by ROM. Standard banking, inherited from prior Spectrum models, selects which 16K ROM is visible in the bottom 16K slot (addresses 0 to 16383) and which 16K RAM bank is visible in the top 16K slot (addresses 49152 to 65535).

The Spectrum +3 introduced a new, so called, **allRAM** mode that could place a limited selection of arrangements of four 16K RAM banks into all four slots. This was not widely used and is often forgotten by programmers who mostly target the 128K Spectrum models prior to the +3. A good example of **allRAM** mode is running CP/M, that requires RAM at the bottom of the address map.

There is a total of four 16K ROMs to select from (inherited from the +3) and a total of **48** 16K RAM banks available (**112** in **2048K** ZX Spectrum Nexts). If you make a quick calculation, that accounts for **832K** in the unexpanded Issue 2 ZX Spectrum Next. The remaining portion of the **1024K** is allocated to other uses, most notably to divMMC memory. The *NextZXOS Startup menu* reports available RAM only, which will be either **768K** or **1792K**.

The Standard banking scheme is controlled by hardware I/O ports (covered in the previous chapter) and via the **BANK** command and its variants which we will examine soon.

The MMU (memory management unit) scheme is diagrammed below. It is much more flexible in that it can map any **8K** bank of physical RAM into any **8K** slot of the CPU's addressable memory.

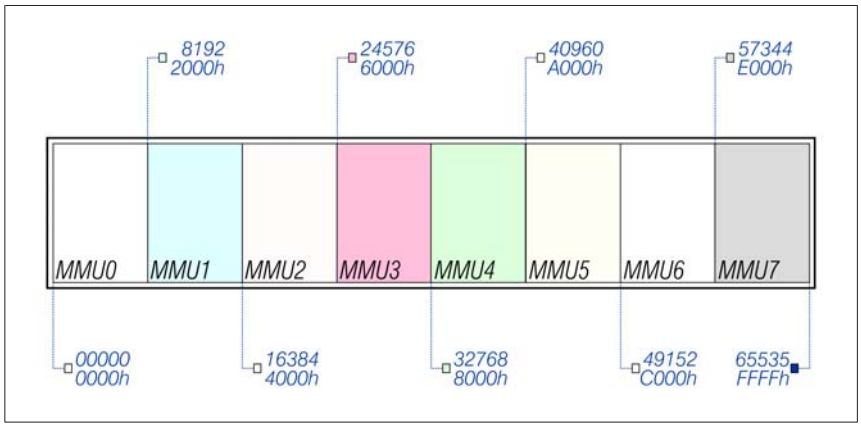



*Fig. 47 – MMU based memory map*

The memory map, is divided into eight slots of 8K named MMU0 through MMU7 and the physical memory is broken into 96 8K banks[1]. Placing a specific 8K bank *n* into the address range 0 to 8191, we might say that 8K bank *n* has been written to MMU0.

Since *NextBASIC* exposes physical memory banks using the Standard scheme's 16K size, we'll concentrate only on this. More information on using the ZX Spectrum Next's MMU system can be found at the end of this chapter, in other sources such as the Spectrum Next *Wiki at* **wiki.specnext.dev** and in *Volume 2 – Advanced ZX Spectrum Next programming* of this manual.

## Reading and Writing to Memory

In the normal course of operations, *NextZXOS* and *NextBASIC* read and write memory on your behalf. As it has been demonstrated in previous chapters, we sometimes need to examine the memory's contents or directly modify it. For these cases *NextBASIC* provides a series of commands and functions to examine and modify memory both in the *memory map* as well as in the whole of the physical memory. These are all variations of two main keywords, namely the **PEEK** (and **PEEK$**) functions (to read the contents of memory) and the **POKE** command (to alter the contents of memory). The full list follows:

| Command | Description |
|---|---|
| PEEK *addr* | Reads the byte at address *addr* |
| POKE *addr,v* | Changes the contents of address *addr* to the byte value *v* |
| DPEEK *addr* | Reads the word stored at addresses starting at *addr* (*addr*, *addr+1*) |
| DPOKE *addr, v* | Changes the contents of addresses starting at *addr* (*addr*, *addr+1*) to contain the 16 bit value *v* |
| PEEK$ (*addr, len/t*) | Reads memory region of length *len* stored in the addresses beginning with *addr* and stores it in a *string* –or–<br>Reads the string terminated with a user specified terminator *t* beginning with address *addr* |
| POKE addr, s | Writes a string *s* in the addresses beginning with *addr* |
| BANK *n* PEEK *o* | Reads the byte at offset *o* in bank *n* |
| BANK *n* POKE *o, v* | Changes the contents in bank *n* at offset *o* to value *v* |
| BANK *n* DPEEK *o* | Reads the word stored in bank *n* at offset *o* (*o*, *o+1*) |
| BANK *n* DPOKE *o, v* | Changes the contents of bank *n* starting at offset *o* (*o*, *o+1*) to contain the 16 bit value *v* |
| BANK *n* PEEK$ (*o,len/t*) | Reads a region of length *len* stored in bank *n* beginning at offset *o* and stores it in a string –or–<br>Reads the string terminated with a user specified terminator *t* from bank *n* beginning at offset o |
| BANK *n* POKE *o, s* | Writes a string *s* in bank *n* beginning at offset *o* |

*Table 22 – PEEK and POKE variants*

As you can see from the table above, *NextBASIC* provides us with a wealth of options to manipulate the contents of both the 64K memory map and the physical memory as a whole. These, complemented by the extended options provided by the **BANK** command, which we will examine further below, can cover almost any memory manipulation need that may arise in the course of writing a program.

Before we continue further with examination of **PEEK**, **PEEK$** and **POKE**, let's first begin with a warning of sorts: Usage of the non **BANK** variants is extremely discouraged. Instead it's best, if you use their **BANK** variants at all times. The reason for that is two-fold and goes back to Memory Banking.

Let's explain; as we said earlier *NextZXOS* and *NextBASIC* update portions of the memory map like the system variables or the display memory if need be. What this means, is that you can't really be sure a value you **POKE**d into the memory map will be there when you try to recover it with **PEEK** unless you take some measures first[2].

Furthermore, **POKE**ing into the memory map unless you absolutely know what you're doing, can have unintended consequences which could result in crashing the machine.

1 *224 in a fully expanded ZX Spectrum Next*
2 *Refer to the CLEAR statement further down this chapter*

We'll first give an example of what could go wrong (it's fortunately safe as an example) and then we'll take a detour and explain how the memory map itself is organised from a *NextBASIC* perspective before returning to PEEK, POKE and their variants. Type:

```
10 POKE 16384,"ABCabc"
20 CLS:a$=PEEK$ (16384,6)
40 PRINT a$
```

From what we've talked about thus far, the intention of the program is obvious (for now also never mind what line 10 does; we'll discuss it later). First we put the word ABCabc into address 16384 of the memory map. Then we try to extract it from the same memory location. RUN the program. What do you see? Certainly not ABCabc you were expecting. Now modify lines 20 and 30 and replace 16384 with 20000 in both lines and RUN the program again.

This is perhaps a contrived example but it shows what happens when you try to use memory that is also being used by something else. In this case, address 16384 is where the contents of the display is stored. After placing the string with POKE in address 16384, a CLS is executed which clears the display and the stored string at the same time.

Here is a trickier example:

```
10 LAYER 1,2
20 POKE 16384,255
30 POKE 24576,255
40 PRINT AT 1,0;"16384 = ";
   PEEK 16384
50 PRINT "24576 = "; PEEK
   24576
```

This program selects *Layer 1,2* (*HiRes*) and then creates two solid and adjacent character sized lines into the display via the POKE commands in lines 20 and 30. Running the program, the results almost seem correct except the character sized line is only one character wide.

The POKE to 24576 did not go to the display in bank 5 because *NextBASIC* placed a different memory page in the memory map to cover the last half of bank 5.

Contrast with the following program that does all its PEEKs and POKEs to bank 5 (the BANK commands will be explained in more detail later). As we will see, PEEK and POKE into a 16K bank, is done using an *offset* into said bank. This means that the "address" range is 0 through 16383; Banks are only 16K long after all. Bank 5, which holds the display is normally placed at address 16384 in the memory map. Performing therefore a POKE into address 16384 is the same as POKE to offset 0 in bank 5. Likewise address 24576 corresponds to offset 8192 in bank 5.

```
10 LAYER 1,2
20 BANK 5 POKE 0,255:
30 BANK 5 POKE 8192,255
40 PRINT AT 1,0;"16384 = ";%
   BANK 5 PEEK 0
50 PRINT "24576 = ";% BANK 5
   PEEK 8192
```

This time, the POKE to 24756 (offset 8192) does go to bank 5 and you will see the solid line twice as wide as the first program.

## NextZXOS and NextBASIC memory allocation

Before we begin to elaborate on *NextZXOS*' memory usage, it should be mentioned that Standard memory management and MMU management are internally synchronised for most cases. Every time a 16K bank is being paged in, the equivalent MMU unit gets the 8K bank component of the larger 16K bank *NextZXOS* uses. As mentioned previously *NextZXOS* also supports **allRAM** mode where the ROM is paged out; this is mainly used by *CP/M.* With this information out of the way, let's see how *NextZXOS* uses the memory.

By default the first 9 RAM banks are used as follows:

| Bank | Description | Address Range |
|---|---|---|
| 0 | Standard 48K Spectrum memory | 49152 – 65535 |
| 1 | RAMdisk | |
| 2 | Standard 48K Spectrum memory | 32768 – 49151 |
| 3 | RAMdisk | |
| 4 | RAMdisk | |
| 5 | Standard 48K Spectrum memory | 16384 – 32767 |
| 6 | RAMdisk | |
| 7 | Used for workspace and data structures by *NextZXOS* | |
| 8 | Used for additional screen data (for *LoRes*, *HiRes* and *HiColour*) and other data by *NextZXOS* | |
| 9 – 111 | Available for user programs (By default banks 9,10 and 11 are used by *Layer 2*) | |

Generally speaking, banks 9+ are always available to the programmer, and can be accessed using the BANK command, while banks 0 – 8 can be used with the following exceptions:

- Bank 0 can be used, only if CLEAR has set the RAMTOP to below 49152.
- Bank 2 can be used, only if CLEAR has set the RAMTOP to below 32768.
- Banks 1,3,4,6 can be used if the BANK 1346 USR command has been used.
- Banks 7 and 8 can never be used.
- Bank 5 can be used with caution.
- Banks 9, 10 and 11 can be used for other purposes if you aren't using *Layer 2* or you have changed their assignments with the LAYER BANK command.

From the above, it is easy to surmise what the initial bank assignments are after boot:

| Slot 1 | Slot 2 | Slot 3 | Slot 4 |
|---|---|---|---|
| ROM | Bank 5 | Bank 2 | Bank 0 |

In case you were thinking that *this looks easy enough – I could page in any bank I want*, don't! In actuality, *NextZXOS* and *NextBASIC* expect certain things to be in certain places at all times within the memory map which is organised in the following manner:

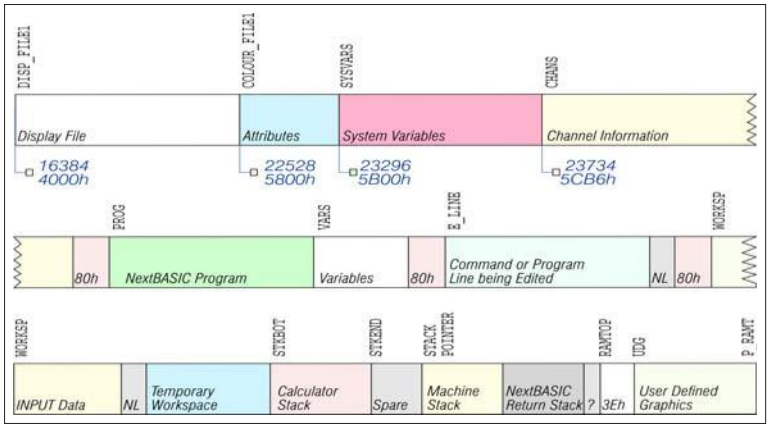



*Fig. 48 – Memory map usage by NextBASIC*

As seen in the figure above, the memory map is divided into different areas that store different kinds of information. The areas are only large enough for the information that they actually contain, and if you insert some more at a given point (for instance by adding a program line or variable) space is made by shifting up everything above that point. Conversely, if you delete information then everything is shifted down. Some areas as you can see include an address below and a name above them whereas others only a name. The areas beginning at an address, signify fixed points in memory such as the *Display and Colour Files*, the *System Variables* and the *Channel Information*. The first three fixed points are required while the fourth (*Channel information*) is an unintended consequence! Let's see why:

The *Display and Colour Files* are as we've seen in previous chapters, legacy areas. The display hardware expects them at these addresses and cannot move inside the memory map. They contain the standard *Layer 0* display memory and parts of *Layer 1* with the rest appearing as needed and managed by *NextZXOS*.

The *System Variables* on the other hand are the system's directory; they contain most information regarding both *NextZXOS* and *NextBASIC* and provide information on the boundaries between the rest of the memory areas on the memory map. In other words following the discussion above, if, say, the last program line changes, it's stored within the area pointed to by the system variable PROG. Some of these locations are marked by the names above the areas in the diagram. A complete list follows in the next chapter. Note, that these are *NextZXOS* variables and not *NextBASIC* variables, so typing these names means nothing to *NextBASIC*.

Now you probably noticed that we said *most information regarding NextZXOS and NextBASIC* and not *all information*. That's because the information that's held in *System Variables* (or SYSVARS) deals with legacy applications and compatibility. *NextZXOS* maintains even more unmovable information elsewhere, tucked away in protected banks and manages it there.

## Memory Areas and their use

Below, let's examine some of the memory areas portrayed in the figure above, as it's helpful to generally know how things are laid out in the memory map.

The *Display and Colour Files areas* store the bitmap for the Layer 0 (and part of the Layer 1) picture. As we saw in chapters *14* through *16*, it is rather curiously laid out, so you probably won't want to PEEK or POKE in it. The upshot of all this is that if you're used to a computer that uses PEEK and POKE on the screen, you'll have to start using SCREEN$ and PRINT AT instead, or PLOT and POINT.

The *System Variables area*, contains various pieces of information that tell the computer what sort of state the computer is in. They are listed fully in the next chapter, but for the moment note that there are some (called CHANS, PROG, VARS, E_LINE and so on) that contain the addresses of the boundaries between the various areas in memory. These are not *NextBASIC* variables, and their names will not be recognised by the computer.

The *Channel Information area* contains information about the input and output devices as seen in *Chapter 20*.

The *NextBASIC Program* and *Variables areas* contain your program and its variables, organised in standard *data structure*s we will examine in the following section.

The calculator is the part of the *NextBASIC* system that deals with arithmetic, and the numbers on which it is operating are held mostly in the *Calculator Stack area*.

The *Spare area* contains the space so far unused.

The *Machine Stack area* is space reserved for the CPU stack.

Similarly, the *NextBASIC return stack area* which was mentioned in *Chapter 4* maintains a record of your program's currently-active subroutine and procedure calls, loops and error handlers.

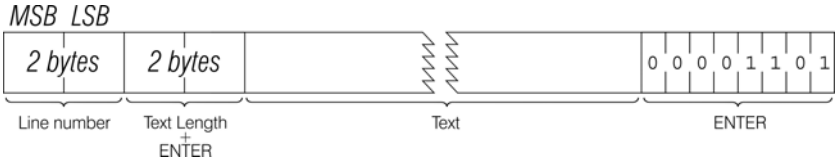
The byte pointed by the RAMTOP variable shows the maximum address that is reserved for use by a *NextBASIC* program. We will visit this in more detail, in the section about the **CLEAR** command below.

Finally the *User Defined Graphics area* holds all the definitions to the system's UDGs as discussed in *Chapter 13*.

## NextBASIC Data Structures

*NextBASIC* stores numbers, strings, arrays, programming lines and **FOR...NEXT** loops in strictly defined forms called *data structures*. The following discuss all these data structures that are user accessible. Integer-based variables, arrays and control structures are not available to the user and are hidden by *NextZXOS* in protected memory areas so they're not covered here.

Each line of *NextBASIC* program has the form:

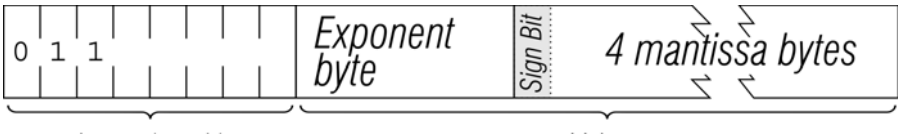



Note that, in contrast with all other cases of *two-byte* numbers in the Z80N, the line number here is stored with its more significant byte (MSB) first: that is to say, in the order that you write them down (also known as *Big-Endian* order).

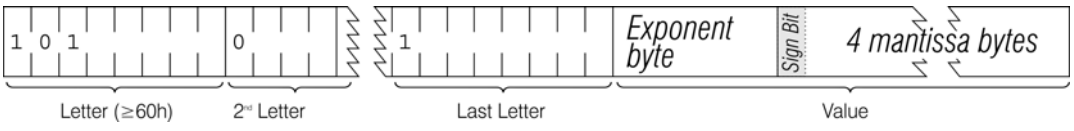
A *numerical constant* in the program appears as ASCII text followed by its binary form, using the character **CHR$ 14** followed by *five bytes* for the number itself.

The variables have different formats according to their features. The letters in the names should be thought as starting off in lower case. The available variants and their formats are:

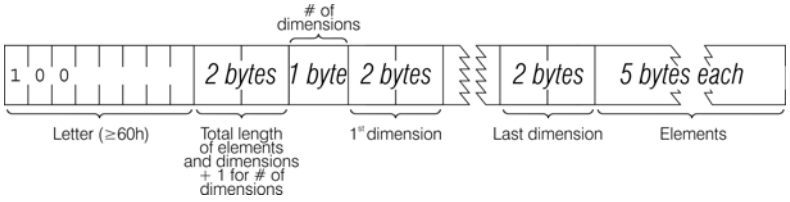
Number whose name is one letter only:



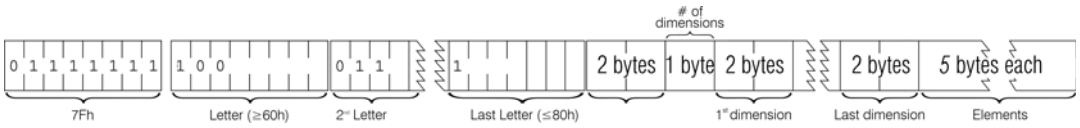

Number whose name is longer than one letter:




Array of numbers:



Array of numbers whose name is longer than one letter:



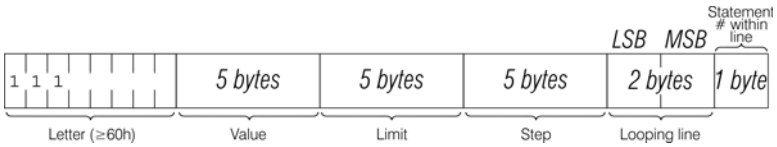

Specifically for arrays the order of the elements is as follows:

- first, the elements for which the first subscript is 1;
- next, the elements for which the first subscript is 2;
- next, the elements for which the first subscript is 3;

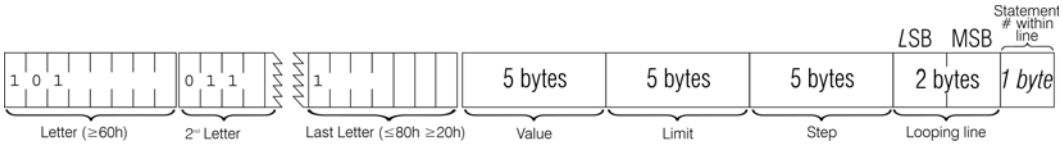
and so on for all possible values of the first subscript.

The elements with a given first subscript are ordered in the same way using the second subscript, and so on down to the last. As an example, the elements of the 3 x 6 array b in *Chapter 11* are stored in the order b(1,1) b(1,2) b(1,3) b(1,4) b(1,5) b(1,6) b(2,1) b(2,2) ... b(2,6) b(3,1) b(3,2) ... b(3,6).

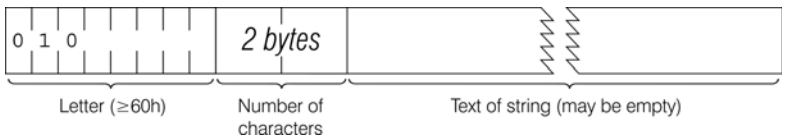
Control variable of a FOR...NEXT loop:



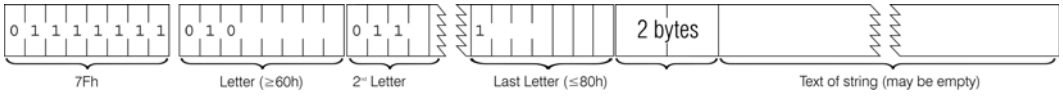

Control variable of a FOR...NEXT loop whose name is longer than one letter:



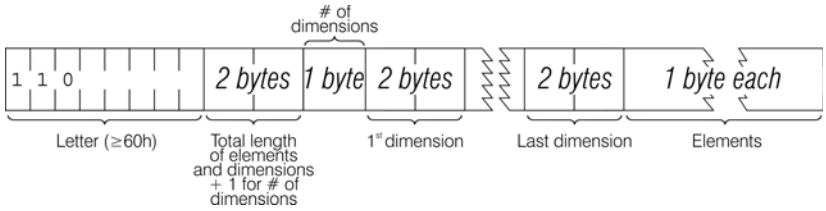

String:



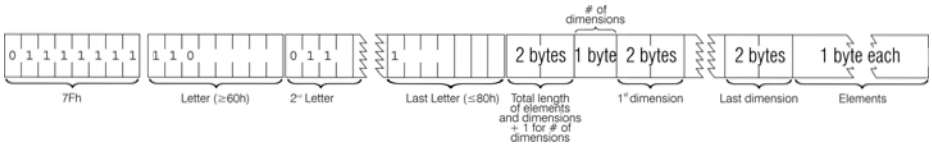

String whose name is longer than one letter:




Array of characters:



Array of characters whose name is longer than one letter:




As you saw in the examples above, numerical values are represented as **5** bytes. These are *floating-point values*. In contrast to integers which are – as discussed in *Chapter 6* and referenced in *Chapters 8 through 11 – of a fixed 16 bit (or two-byte) size, floating-point* numbers can represent both decimal *and* integer values. Due to the calculations involved, their usage will slow down your programs; so avoid using them if you do not need decimal points or values higher than **65535**.

For *floating-point* values, *any* number (except **0**) can be written uniquely as: $\pm m \times 2^e$

where $\pm$ is the sign, *m* is the mantissa, which lies between ½ and **1** (it *cannot* be **1**), and *e* is a *biased exponent*.

Suppose you write the fractional *m* in binary. Because it is a fraction, it will have a binary point (like the decimal point in decimal) and then a binary fraction (like a decimal fraction). So in binary:

| | | |
|---|---|---|
| one half | is written as | .1 |
| one quarter | is written as | .01 |
| three quarters | is written as | .11 |
| one tenth | is written as | .00011001100110011001 |

and so on.

With our number *m*, because it is l*ess than* **1**, there are no bits before the binary point, and because it is *at least* ½, the bit immediately after the binary point is a **1**. To store the number in the computer, we use *five bytes*, as follows:

I. write the *first eight* bits of the *mantissa* in the *second byte* (we know that the first bit is **1**), the *second eight* bits in the *third byte*, the *third eight* bits in the *fourth byte* and the *fourth eight bits* in the *fifth byte*
II. replace the *first* bit in the second byte which we know is **1** by the sign: **0** for plus, **1** for minus
III. write the *exponent* +**128** in the first byte.

For instance, suppose our number is $^1/_{10}$:

$^1/_{10} = {}^4/_5 \times 2^{-3}$

Thus the mantissa *m* is **.11001100110011001100110011001100** in binary (since the *33rd* bit is **1**, we shall round the *32nd* up from **0** to **1**), and the exponent *e* is **-3**.

Applying our three rules gives the *five bytes*:

There is an alternate way of storing whole numbers between **-65535** and **+65535**:

I. the *first* byte is **0**
II. the *second* byte is **0** for a positive number, **FFh** for a negative one
III. the *third* and *fourth* bytes are the less and more significant bytes of the number (or the number **+131072** if it is negative),
IV. the *fifth* byte is **0**.

This is essentially the *two's complement* representation we discussed in *Chapter 6* for integers with two extra bytes, one before and one after the number and an entire byte dedicated to the sign as opposed to one bit only. Compared to the integer type supplied by the Integer Expressions evaluator, it is wasteful memory-wise and slower to process.

## PEEK, POKE and their variants

Now that we've examined more thoroughly what the memory map looks like to *NextBASIC*, it's time to revisit the commands and functions that read and modify its contents.

To inspect the contents of one or more memory locations, we use the PEEK, DPEEK and PEEK$() functions; The PEEK variant functions are always safe to use as they change nothing in memory; they can however give unpredictable results in cases where a memory location is marked for moving. As we saw however, there are places in memory which are unmovable; reading in the System Variables area for example is a always a predictable scenario. For instance, this program prints out the first *21* bytes in ROM (and their addresses):

```
10 PRINT "Address"; TAB 8; "Byte"
20 FOR a=0 TO 20
30 PRINT a; TAB 8; PEEK a
40 NEXT a
```

All these bytes will probably be quite meaningless to you, but the processor understands them to be instructions telling it what to do.

DPEEK is similar but since it returns 16 bit values, the example above would have to be rewritten as follows:

```
10 PRINT "Address"; TAB 8; "Word"
20 FOR %a=0 TO 20 STEP 2
30 PRINT %a; TAB 8; DPEEK %a
40 NEXT %a
```

Generally speaking, (D)PEEKing into ROM is very much useless and it's much more likely that you'll use (D)PEEK to either read a system variable or read a value you've previously POKEd. (D)PEEK$ () on the other hand returns the values at an address in memory in the form of a string. Its syntax is as follows:

PEEK$ (*address*, *argument*)

where *address* is any address in the memory map, while *argument* can be one of the following:

1. A number signifying a length of characters to be retrieved
2. A single tilde ~ character, to find any bit-7 terminated string (that means that bit-7 of the last character in the string is set)
3. A tilde ~ character followed by the ASCII code of one character that terminates the string

Let's look at an example which helps us search in memory (albeit very slowly):

```
10 RUN AT 3: REM this takes a
   long time!
20 FOR %a=0 TO 65535
30 PRINT AT 0,0;"Now scanning
   address:";%a
40 a$= PEEK$ (%a,8)
```

```
 50 IF a$="Variable" THEN PRINT
    AT 1,0;"Found word at
    address:";%a: GO TO 70: REM
    stop iterating here and go
    below
 60 NEXT %a
 70 FOR %a=0 TO 65535
 80 PRINT AT 3,0: "Now scanning
    address:";%a
 90 a$ = PEEK$ (%a, ^101)
100 IF a$="Variabl" THEN PRINT AT
    4,0; "Found word at address ";%a
110 NEXT %a
```

You'll undoubtedly notice that line 100 says Variabl instead of Variable and that's because the terminator character we set in like 90 to look for, is not included in the string returned by PEEK$(). What this program actually finds is the address in the memory map where line 50 is stored! The second half of this example (lines 70 on) is very much pointless but was made to show the flexibility of PEEK$()'s arbitrary termination character search.

Normally, it is much more likely to read for NUL terminated strings (~0), FFh terminated strings (~255), often used in +3DOS/IDEDOS and perhaps CR terminated strings (~13), if for example the data you're searching for has been PRINTed with line separators.

To change the contents of a RAM address in the memory map, we use the POKE or DPOKE statements. These have the form:

POKE *address*, *value1[,value2[,value3...[,valueN]]]*
DPOKE *address*, *value1[,value2[,value3...[,valueN]]]*

The ability to POKE gives you immense power over the computer if you know how to wield it; and immense destructive possibilities if you don't. It is very easy, by poking the wrong value in the wrong address, to lose vast programs that took you hours to type in. Fortunately, you won't do the computer any permanent damage.

As we mentioned earlier, POKE is generally not safe to use within the confines of the memory map, unless you either know what you're doing, or the area you're modifying is fixed (like say the *Layer 0* screen or attribute areas or the System Variables – the latter always with caution). It's also safe to POKE within the memory map if you have used the CLEAR command and modify the area above it.

Let's try modifying a *system variable* to show how powerful POKEing can be:

First, type test in the editor and once you hit ENTER your computer will complain with a buzzing sound. The variable that holds let length of that buzz is called RASP and it's located in address 23608 (5C38h) within the System Variables area.

Now, let's see how can we adjust that buzz. We'll start by looking what is its current value with:

```
PRINT PEEK 23608
```

Then modify it with

```
POKE 23608, 16
```

Type **test** again and press **ENTER**. The buzz indicating the error in your code, shortened in length. You can experiment with different values. The new value you enter must be between **-255** and **+255**, and if it is negative then **256** is added to it.

**POKE** is not confined into a simple byte sized value as you may have surmised. In fact it can accept a mix of numbers and strings, in a comma separated list of values with each accepting an optional tilde ~ character suffix. In the case of numeric values, the optional tilde suffix after each value makes that value 16 bits wide (a word) while in the case of strings, the optional tilde suffix sets the most significant bit of the last character in the string, usually known as *bit7-termination*. This is sometimes used in order to store variable-length strings in a compact way. However, it's usually more convenient to use a byte such **0** (**NUL**) (*null-termination*), **255** (**FFh**) or **13** (**0Ch**) (**CR**) to terminate a variable-length string in memory. Note that for **DPOKE**, the tilde is used after numeric values to specify values that should be written to memory as a byte rather than a 16 bit word. You can therefore think of the tilde as a *write this value in the opposite way to the default for this command* designator! Let's see a few examples:

```
POKE 32768,200
```

modifies the contents of the byte address **32768** to **200**.

```
POKE 32768,8,9,10,"test",30000~,55
```

modifies the contents of address **32768** to **8**, address **32769** to **9**, address **32770** to **10**, addresses **32771** to **32774** to contain the string **test** (or in other words the values **116**, **101** ,**115** and **116** respectively – the ASCII codes for the letters making up the word **test**), addresses **32775** and **32776** to contain values **48**, **117** respectively (or 117 x 256 + 48 = 30000) and finally address **32777** to **55**. In other words we **POKE**d 3 bytes, a string, a word and a byte.

```
DPOKE 32768,1000,2000,3000,100~,2
```

modifies addresses **32768** though **32775** (pokes 3 words, a byte –with the tilde–and a word).

In *Chapter 13* we briefly discussed **POKE USR** "*letter*" . That may look like a separate variant of **POKE** but in reality **USR** "*letter*" is just as shortcut to the address of the UDG defined by *letter*. There is a small caveat that when used in a single value context, 8 successive POKE USR commands must be given (one for each row in the 8x8 matrix of the UDG) so it's always better if we use it in a list of values context, like so:

```
POKE USR "A",1,3,7,15,31,63,127,255
```

which redefines UDG **A**.

Using **POKE** with strings is equally powerful so it deserves a separate example. Let's use the example that used **PEEK$()** to search for a string in order to demonstrate a bit of *NextBASIC* memory areas magic! First delete all lines after 60 and modify line 20 to read:

```
20 %a=22000 TO 65535
```

(This change is to make sure the program doesn't take forever).

**RUN** the program and when you find the address, note it down, then do the following:

```
POKE address, "Horrible"
```

where `address` is the address you noted earlier. Press **ENTER**, then write **LIST** and look at line 50. See? Magic!

Note that using the first form of **POKE** to any address between **0** and **16383** (the ROM slot) will have no effect regardless of what you attempt to do as shown by this example:

```
FOR %f=0 TO 16383: POKE %f,0: NEXT %f
```

The same however is not entirely accurate for DPOKE and the string POKE version of the command. For example both the commands that follow will NOT write in the ROM slot but WILL write in the RAM slot (it so happens as you see from the previous figure) that the first area right after the ROM is DISP_FILE so you'll see a visual result immediately:

```
POKE 16383, "This is a test":PAUSE 0
```

and

```
DPOKE 16383, 65535: PAUSE 0
```

will both produce a visible result in the upper left corner of the display while the ROM slot is not affected.

## CLEAR

When looking at the different memory areas maintained by *NextBASIC*, we briefly mentioned the System Variable RAMTOP. This variable (located at address 23730) contains the address of the last byte used by *NextBASIC*. Even NEW, which clears the RAM out, only does so as far as this address – so it doesn't change the user-defined graphics. You can change the address RAMTOP points to by putting it as an numeric argument in a CLEAR statement as follows:

CLEAR *new_RAMTOP*

This effectively does 4 things:

- clears out all the variables
- clears the display file (like CLS)
- does RESTORE
- clears the *NextBASIC return stack* and puts it at the *new_RAMTOP* address – assuming that this lies between the calculator stack and the physical end of RAM; otherwise it leaves RAMTOP as it was.

RUN also performs a CLEAR, although it never changes RAMTOP.

Using CLEAR in this way, you can either move RAMTOP up to make more room for *Next BASIC* by overwriting the user-defined graphics, or you can move it down to make more RAM that is preserved from NEW. It can also be used to ensure that the machine stack is below BFE0h (49120) when intending to call *NextZXOS* – this means that the stack will not have to be subsequently moved within your own machine code.

Type NEW, select *NextBASIC*, then CLEAR 23800 to get some idea of what happens to the machine when it fills up. You'll immediately get an M RAMTOP no good error message. Trying CLEAR 23900 will report 0 OK but attempting to write a program will stop with a buzzing sound very quickly. That means that the *NextBASIC* user program memory is now full and you will have to make room before typing any more. There are also two error messages with roughly the same meaning, 4 Out of memory and G No room for line.

It's worth mentioning that the *Clear option* in the *NextBASIC menu* (accessible by pressing the EDIT key) can also be used to CLEAR memory and it's particularly useful if you have cleared RAMTOP too low and no longer have enough memory to enter *NextBASIC* commands as with the example above. It sets RAMTOP to just below the current UDG area (ie. equivalent to CLEAR % DPEEK 23675-1, one less than the value in the UDG SysVar).

## Memory Bank management with BANK

Under *NextBASIC* the system's memory capacity is shown in the on-screen menus. It can also be queried programmatically by examining the new system variable, MAXBNK, which contains the number of the highest usable bank in the system (normally 47 or 111)[3].

3 *The dot command* ***.mem*** *also returns the memory information, although measured in 8K banks.*

To make all the extra memory easily accessible to the user, *NextBASIC* provides a special command called BANK which can be combined with a number of normal commands to extend their functionality to the whole of the ZX Spectrum Next's memory and not just the memory map addresses. We've seen some already used in the course of this guide, especially in chapters *14* through *17 as well as Chapter 20*.

Memory banks are marked as *in-use* or *free* by the user or by commands that access them (BANK … PEEK / PEEK$ / POKE / COPY / ERASE / USR / LAYER, LAYER … BANK and LOAD … BANK). Users can mark a bank as *in-use* or as *free*, by either using an explicit command from the list above or one of the two special commands BANK NEW *var* and BANK *n* CLEAR.

BANK NEW *var*

Reserves the next available free bank number and assigns it to the numeric variable *var*, ready for use with and by other BANK commands. This command is useful for allocating banks for use in *NextBASIC*, allowing for cases where a resident machine code program has previously allocated banks for its own use.

Note, that is not essential to use this command, as commands such as LOAD … BANK will automatically allocate the specified bank for use by *NextBASIC*, but only if the specified bank is not already in use by a resident machine code program.

Let's try a small example. Assuming you have a 2048K ZX Spectrum Next; type the following program:

```
10 FOR %f = 0 TO 111
20 BANK NEW a
30 PRINT AT 0,0; "Allocating
   bank:"; a
40 NEXT %f
```

Once you RUN it, the program will begin to allocate memory banks and print the ones it allocates; you'll notice two things: Allocation begins at bank 111 (47 if using an unexpanded KS1/Issue 2 ZX Spectrum Next) and progresses backwards and that program execution will stop abruptly with a 4 Out of memory error report once you reach a bank that's allocated by the system as described in the *NextZXOS and NextBASIC Memory Allocation section*. Indeed, if you use the dot command .mem then you'll see that you have 0 banks free (0K). In order to free up a bank to be used, you will need to use the BANK *n* CLEAR command whose syntax is as follows:

BANK *n* CLEAR

Marks bank *n* as *free* for use by other parts of the system (eg dot commands).

Let's try to free a bit of memory after the mess we've made with the previous program. Without making any more changes, let's try:

```
BANK 11 CLEAR
```

More likely than not, the system will report: In Use, 0:1. What has happened? Most likely that the bank itself is in use by the system. Let's try again:

```
BANK 12 CLEAR
```

This time the system will most likely report: 0 OK, 0:1. We can verify this by running .mem again. This time it will show us 2 Banks free (Remember .mem reports memory in MMU sized banks – that is 8K). Bank 11 you tried to free originally (unless the system hasn't been modified), is being used by *Layer 2 (*which takes 3 banks, *by default* 9,10 and 11but can be changed by the LAYER...BANK command *)* so it's rightfully marked as *in-use*. Note

here that if you're not using *Layer 2*, the banks it occupies CAN be used for other purposes including machine code programs. They just cannot be released.

Banks marked as *in-use*, remain reserved after a NEW command and are only released at a reset (or with this BANK *n* CLEAR).

BANK CLEAR reports A Invalid Argument, 0:1 if you try to clear banks 1,3,4 and 6 even if you have given the BANK 1346 USR command which is described below.

*NextZXOS* allocates 64K to the RAMdisk by reserving banks 1,3,4 and 6; BANK 1346 USR allows you to release these for use by your programs. Once you give the command:

```
BANK 1346 USR
```

the following things happen; first all files in the RAMdisk are deleted, then the drive itself is unmounted and using BANK commands on these banks stops producing errors. To undo this action and reinstate the RAMdisk you will need to use:

```
BANK 1346 FORMAT
```

which will erase the contents of these banks and re-attach them to the RAMdisk. The disk itself however will need to be manually mounted again by using the MOVE...IN command. See *Chapter 19 for details.*

Bank contents can be copied and erased in whole or in part using the BANK COPY and BANK ERASE commands.There's also a specific one that copies data quickly to and from the screen but we'll look at that separately. The syntax to copy bank data is:

BANK *source_bank* COPY [*source_offset, len*] TO *destination_bank [,dest_offset*]

where *source_bank* is a readable bank number to copy *from* while *destination_bank* is a writeable bank number. *Source_offset* and *len* signify the location within the source bank and the size *in bytes* of the memory chunk we're copying. If the latter are specified, then the *dest_offset* must also be specified. Let's try:

```
BANK 9 COPY TO 47
```

will copy the bank holding the first third of *Layer 2* into bank 47 while,

```
BANK 1 COPY TO 47
```

will return A Invalid argument, unless BANK 1346 USR has been used!

```
BANK 9 COPY 8192, 8192 TO 47, 0
```

will copy the bottom half of the first third of the *Layer 2* screen to the start of bank 47 (Once you untangle that tongue-twister you can see how this can create interesting blinds effects!).

It's also quite handy to quickly erase the whole or part of a bank (fill it with zeroes or an arbitrary byte value). This is accomplished by the BANK ERASE command whose syntax is:

BANK *n* ERASE *[offset, len][,][value]*

where *n* is the number of writeable bank, *offset* is the optional starting point of the erase and *len* is the length (in bytes) of the area to be erased. The optional *value* will fill the area with a byte of your choosing or – if omitted – 00h. Here are some examples using *Layer 2* and an image present in your System/Next™ distribution (you will need to provide the image in 256 x 192 x 256 colour BMP format):

```
10 CD "ENTER HERE THE FOLDER"
20 LAYER 2,1
30 .bmpload yourfile.bmp
```

```
 40 BANK 9 COPY TO 111: REM
    first we copy it
 50 PAUSE 0: REM wait for a
    key
 60 BANK 9 ERASE 128:; Erase it with
    value 128 which is by default a
    red colour for Layer 2
 70 PAUSE 0:; wait for a key
 90 BANK 111 COPY TO 9:; restore it
100 PAUSE 0:; wait for a key
110 LAYER 2,0: LAYER 0
```

You can see easily how fast this happens (and how it can be used for a myriad of applications)

## Using BANK with graphics

Over the course of chapters dealing with graphics, we've used a lot of graphics-related commands that involved the use of BANK. These are BANK LAYER, LAYER BANK, LAYER PALETTE BANK, SPRITE PALETTE BANK, SPRITE BANK, TILE BANK and TILE DIM all benefiting all providing significant speed enhancements both in development and in usage.

We saw above the use of BANK COPY to copy data from one bank to another. This includes Layer data as they too are kept in banks and managed by *NextZXOS*. There is however a specially crafted command that does this and more as it adds more options specifically tuned to the requirements of display. Unlike BANK COPY, this is designed to update small areas of the screen to facilitate effects and especially animation. The command is BANK LAYER and it is used to quickly copy data from a memory bank to the *screen in the current mode*, or vice versa. The syntax is as follows:

BANK *n* LAYER *x,y,w,h*|*offset* TO [*raster_op*] *offset*|*x,y,w,h*

where *n* is the source OR destination bank number, *x* and *y* is the top left character position expressed in character column and row coordinates, *w*, *h* are the width and height again in characters of the area to be *copied from* or *copied to*, *offset* is the starting offset in the bank we'll be copying to or from while *raster_op*, is an optional symbol modifier to TO that affects the data being copied *at their destination* (does not affect the source data).

TO *raster_op* can be one of the following values:

| | |
|---|---|
| TO | Straightforward copy |
| TO & | ANDs the copied data onto the destination |
| TO \| | ORs the copied data onto the destination |
| TO ^ | XORs the copied data into the destination |
| TO ~ | Copies data into the destination unless it is equal to the global transparency colour (default E3h); if so, leaves the destination unchanged |

The area of screen copied by BANK...LAYER is defined as with Windows in characters. That means that character positions range from 0 to 31 for *x* and 0 to 23 for *y*, for all modes *except LoRes*, where they range from 0 to 15 for *x* and 0 to 11 for *y*.

Data copied from the screen is laid out as follows, depending upon the currently selected layer (see *Chapter 16*):

### Standard resolution (Layers 0 and 1,1)

The attribute data comes first, stored as *h* consecutive rows of attributes, *w* bytes wide. Following this is the screen data, stored as *h* × *8* consecutive rows of pixel data, *w* bytes wide. The total memory used is therefore **w** × **h** × **9** bytes.

### HiRes (Layer 1,2)

In this mode, each character position is 16 pixels wide, comprising a left and right "half". The screen data is stored as *h* × 8 consecutive pixel rows of data. For each row, the first *w* bytes comprise the left halves of all characters. The next *w* bytes in the row comprise the right halves of all the characters. The total memory used is therefore **w** × **h** × **16** bytes.

### HiColour (Layer 1,3)

The screen data is stored as *h* × *8* consecutive pixel rows of data. For each row, the first *w* bytes comprise the pixel data. The next *w* bytes in the row comprise the attribute data. The total memory used is therefore **w** × **h** × **16** bytes.

### LoRes (Layer 1,0), Layer 2 standard

The data is stored as *h* × *8* consecutive pixel rows of data. For each row, there are *w* × *8* bytes, with each byte representing a single pixel. The total memory used is therefore **w** × **h** × **64** bytes.

In the previous section, we dealt with bank management. The following command could very well belong there, but since it deals with memory management of the graphics subsystem and specifically with *Layer 2*, we will cover it here. **LAYER BANK** redefines which banks will store Layer 2 display data (the *front buffer*) and which will act as the *back buffer* (for rendering). The syntax is as follows:

**LAYER BANK** *n,m*

where *n* is the front buffer base bank number for *Layer 2* (this also sets *n+1* and *n+2*) and *m* is the back buffer base bank number (and also sets *m+1* and *m+2*). These values can be the same and both default to **9**. Unlike other **LAYER** commands, it can be executed in any mode. For example to move *Layer 2* to banks **13** to **15** (front buffer) and **16** to **18** (back buffer):

```
LAYER BANK 13,16
```

If we now give:

```
BANK 9 CLEAR
```

We can see that bank **9** (the original base bank for *Layer 2*) can now be released. The effects of **LAYER BANK** can be undone either by reversing the command, with **NEW** or with **LAYER CLEAR**.

Memory banks are also ideal to store palette information as palettes are basically a series of 256 bytes or words (depending on your **PALETTE DIM** setting). There are two commands for that: **LAYER PALETTE BANK** and **SPRITE PALETTE BANK**. Their syntax is virtually identical and is as follows:

**LAYER|SPRITE PALETTE** *n* **BANK** *b,offset*

where *n* is the palette number (**0** or **1**), *b* is the bank number and *offset* is the start location in the bank where the palette values are located. As mentioned above, if **PALETTE DIM** was set to **8**, **LAYER** and **SPRITE PALETTE BANK** will load **256** bytes from bank *b*, *offset*, while if **PALETTE DIM** was set to **9**, **512** bytes will be loaded.

Apart from the palettes, sprite definitions[4] themselves can be stored and exchanged through the use of memory banks. The command and its syntax to define either all 64 sprites at once (64 sprites of 256 bytes each equals a full bank of 16K) or some of them is:

SPRITE BANK *b [, offset, pattern_no, number_of_sprites]*

where *b* is the bank number holding the sprite pattern definitions, *offset* is the starting location in the bank where sprite definitions are stored, *pattern_no* is the starting pattern number that's defined by the command and *number_of_sprites* is the total number of sprites that are defined. If we store all 64 sprite definitions within a bank, then the command can be as simple as:

```
SPRITE BANK 14
```

which will load 64 sprite definitions from bank 14. Alternatively to load 32 sprite definitions starting with pattern number 4 from bank 15 offset 256 would require:

```
SPRITE BANK 15,4,256
```

Sprites and tiles (not to be confused with *Layer 3 tiles*) are closely related. As a matter of fact as we saw in *Chapter 17, their main difference is that tiles are managed by software and not hardware, so it follows that NextBASIC* provides similar commands to manage them at least memory-definition wise. The BANK commands related to tiles are TILE BANK to define the tiles themselves and TILE DIM to define the tilemap, that is how are the tile patterns organised. The syntax of the first is:

TILE BANK *n*

where *n* is the number of the base bank holding the tiles. If more are needed as defined by the tilemap, they will be taken from subsequent bank numbers (up to an additional 3 making a total of 4 banks assigned to tile definitions). The tilemap itself is also held in a bank and managed with:

TILE DIM *n*,*offset*, *w*, *tile_size*

which defines the tilemap in bank *n*, starting at location *offset* with width *w* which ranges from 1 to 2048 and tile size *tile_size* (8 for $8 \times 8$ pixels or 16 for $16 \times 16$ pixels).

## Using BANK with files

The entire range of BANK commands for file management, has been covered in length throughout *Chapter 21 – NextZXOS and alternatives* so we'll just include them here for completeness and as a quick reference. As a general guideline for syntax, BANK does not need an offset and length for SAVE operations except the ones that deal with fixed areas. The commands that deal with files and their syntax are:

LOAD|SAVE|VERIFY *filespec* BANK *n* [,*offset,length*]

and the additional

SAVE|LOAD *filespec* LAYER

that are special shortcut commands to load and save the current layer display. This obviously includes bank access (as for example *Layer 2* occupies 3 banks) and thus it's included here. In all the above, *filespec* is a valid filespec for the filesystem you're accessing, *n* is the bank number while the optional *offset* and *length* must be given together to signify the starting location and length of the data chunk we're manipulating. If omitted the entirety of the bank is used.

4 *Although the ZX Spectrum Next's Sprite Engine can define and manipulate a total of 128 sprites, these only work with 4 bit palette definitions which are not supported by NextBASIC. Instead NextBASIC supports a total of 64 sprites of 256 colours each*

## Extending NextBASIC Programs with BANK

Unlike previous iterations of Sinclair BASIC, *NextBASIC* makes it possible to write programs larger than the approximate 41K which used to be the norm with previous ZX Spectrum models. This is achieved through the use of BANK command extensions; whole sections of *NextBASIC* programs can be copied into any memory bank available to the user (and saved/loaded with the SAVE / LOAD...BANK commands as described in *Chapter 20* as well as the previous section). Programs can then switch between lines in the "main" program area and those held in a bank.

The following new commands are available to manage banked sections of *NextBASIC* programs: BANK LINE, BANK LIST and BANK LIST PROC(), BANK MERGE, BANK GO TO, BANK GOSUB, BANK PROC and BANK RESTORE. We have covered these as well in the appropriate sections of this guide, so they're mentioned here in brief for completeness and reference. Syntax is as follows

BANK *n* LINE *x,y*

Copies lines *x* through *y* (inclusive) from the main program to bank *n*. The total number of bytes used in the bank will be shown. Once this has been done, it is not possible to change or delete any lines in the banked section, except by completely overwriting the bank's contents using another BANK...LINE command or by executing a command that will replace the bank's contents with something else.

BANK *n* LIST [*l* | PROC *name*()]

Lists lines, optionally starting with line or label *l* or from a procedure named *name*, in bank *n*.

BANK *n* MERGE

Copy all lines back from bank *n* into the main program. This won't overwrite line numbers that did not exist in the source bank

BANK *n* GO TO *l*

performs a GO TO line or label *l* in bank *n*. To GO TO to a line or label in the main program from a banked section, the bank number should be 255.

BANK *n* GOSUB *l*

branches using GOSUB to the subroutine located at line or label *l* in bank *n*. To GOSUB to a subroutine in the main program from a banked section, as with GO TO above, the bank number should be 255.

BANK *n* PROC *name* (*parameter1*[*,...,parameterN]*)[TO *variable1*[*,...,variableN*]]

branches to the PROC named *name* located in bank *n* with optional parameters *parameter1* to *parameterN* and optional return values stored in *variable1* to *variableN*. To branch to a PROC in the main program from a banked section, as with GO TO above, the bank number should be 255.

BANK *n* RESTORE *l*

Sets the DATA pointer to line or label *l* in bank *n* ready for the next READ operation.

It's noted that BANK LINE and BANK MERGE can only be given as direct commands and not as part of a saved program be it in a bank or in the main section.

## NextZXOS Paging Mechanism Overview

As we discussed in the introduction to this chapter the CPUs used in all previous models of the ZX Spectrum line as well as this one, can only address 65536 bytes. The original 128K ZX Spectrum crammed in more than twice the amount of memory than it could address clocking in at 131072 bytes of RAM and 32768 bytes of ROM making 163840 bytes

(160K) in all. The +3 that followed it a few years later increased that to almost 192K with an additional 32K of ROM while the Next has increased that number even further to 1024K or 2048K depending on if you have expanded the ram on your machine or not.

All the extra memory is hidden from the processor by the hardware using a process called paging – *NextBASIC* (and the processor) always *sees* the memory as 16K of ROM and 48K of RAM (or 64K of RAM with no ROM in allRAM mode – though that is never used by *NextBASIC* and *NextZXOS* and it's reserved for CP/M).

While the processor can indeed address only 64K of memory at once, the extra memory can be slotted in and out of that 64K at will as seen in the introduction to this chapter. Consider an old jukebox. Although it (and you) can only deal with one album at a time, there are many more albums there which can be selected with the right buttons. So, even though there's much more information than you can use at any one time, you can pick and choose which part is relevant.

It is much the same for the processor. By setting the right bits in an I/O port, it can pick and choose which chunks of the available of memory it wants to use. When in non–banked usage of *NextBASIC* as well as when using legacy software most of the memory is ignored, but for Next mode games playing, *Layer 2* graphics and the use of all the new capabilities the ZX Spectrum Next is equipped with, having sixteen or even thirty two times as much RAM is really rather useful!

Normally, usage of the additional memory capabilities are handled directly by *NextZXOS* and *NextBASIC* either automatically or by using the BANK commands, however in order to understand the underlying mechanisms we can elaborate a little bit.

Look again at the memory map; RAM pages 2 and 5 are always in the positions shown when *NextBASIC* is used, though there's no reason why they shouldn't be in the "legacy banked" section (C000h to FFFFh) – however, it would be difficult to see any use for this.

For legacy usage (usually where programs generate very strictly timed video effects), RAM banks are considered as being of one of two types: contended (meaning that there's a competition between the CPU and the ULA for access to them) and uncontended (meaning the CPU has their exclusive use).

Only four banks are ever contended: banks 4 to 7. The rest of the available RAM banks are always uncontended. This is a setting that can be turned on an off by using a Next Register as we saw in the previous chapter. It's turned OFF by default, but for compatibility reasons, *NextZXOS* turns it ON when loading software in a legacy format (.SNA, .Z80 or .TAP). When writing software that may be used in older models, place any machine code which has critical timing loops (such as music) in uncontended banks[5].

Assuming contention has been turned ON, to turn it OFF you will need to issue a:

```
REG 8, % REG 8 | %01000000
```

command, setting therefore NextREG 8, D6 to 1. The inverse (setting it to 0) will turn contention ON again for these banks. Alternatively you can just press the NMI button and set it/reset it using the *NMI menu* under *Settings > General* which is much much easier!

The ZX Spectrum Next uses a combination of paging techniques we called *standard* at the beginning of this chapter. In reality, it uses three: The 128K style paging (described below) controlled by I/O address 7FFDh, the +3 style paging controlled by I/O address 1FFDh extended by Next Memory Bank Select control controlled by I/O address DFFDh.

The reason for this complicated scheme is that the original ZX Spectrum 128K which introduced banking, only had 8 pages of RAM (8 × 16K) to deal with and only two of ROM (2 × 16K) so there was no appropriate care taken for further expansion. In an original 128K ma-

5 *For comparison, executing NOPs in contended RAM will give an effective clock frequency of approximately 2.6MHz as opposed to the normal 3.5MHz in uncontended RAM for the base clock speed. This is a speed reduction of about 25%*

chine only the top slot (slot 4) of the address space was banked in and out by the user (located at address range C000h to FFFFh.

When the ZX Spectrum +3 came out, there were two more 16K ROMs introduced, which didn't originally exist; that paired with the need to run CP/M that requires RAM at the bottom of the address map, necessitated the creation of yet another I/O address: 1FFDh.

Between these two ports, there are enough bits to address all the RAM pages of an unexpanded Next, however, on a fully expanded Next, one more port was needed to be able to address the entire physical memory available. These methods are all extending one another so backwards compatibility is ensured, while the introduction of the MMUs allows for a more straightforward memory management system for user programs.

Let's begin how this all works by first looking at 128K style paging. The hardware port that

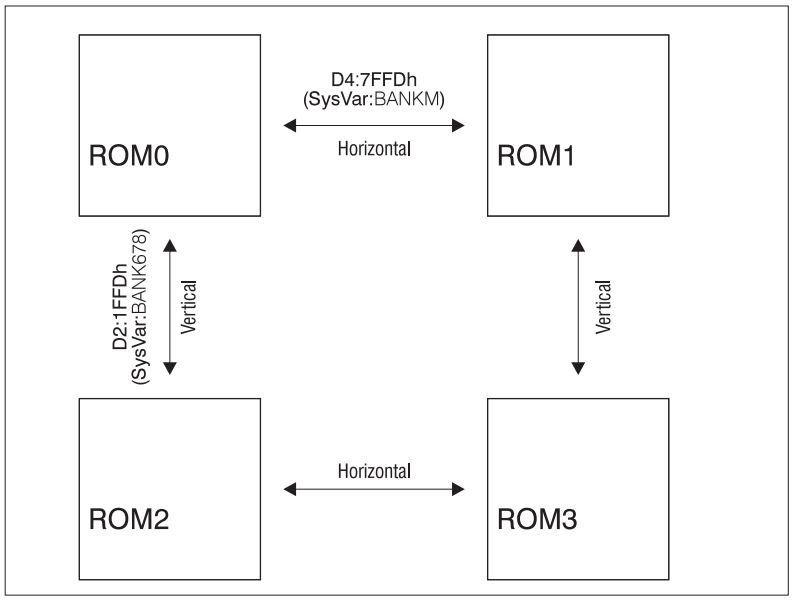



*Fig. 49 – Horizontal vs Vertical ROM switching*

controls it, is at I/O address 7FFDh (32765). The bit layout for this port is as follows:

| Bit | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| Description | | | Disable Paging | ROM Select | Screen Select | RAM Select | | |

D2 to D0 is a three bit number that selects which RAM page goes into the C000h to FFFFh slot. In previous models (such as the +3e) in BASIC, RAM page 0 was normally in-situ, and when editing, RAM page 7 was paged in for various buffers and *scratchpads*.

D3 switches screens: Screen 0 (the Display + Colour Files) was held in RAM5 (beginning at 4000h) and it was the one that BASIC used, screen 1 was held in RAM7 (beginning at C000h) and could only be used by machine code programs.

D4 determines whether ROM0 (the editor ROM) or ROM1 (the 48K BASIC ROM) is paged into Slot 1 at 0000h to 3FFFh.

D5 is a safety feature – once this bit is set, no further paging operations will work. This is normally used when the machine assumes a standard 48K Spectrum configuration and all the memory paging circuitry is locked out. On previous models, this meant that it couldn't be turned back into a 128K machine other than by rebooting; however, the sound chip can still be driven by OUT either from 48K Basic or machine code. On the ZX Spectrum Next however, you can override that lock switch it back to on by setting NextREG 8, D7 to 1.

Note here that the **16K** Bank **5**, is the bank read by the ULA to determine what to show on screen for *Layer 0* (and 1). The ULA connects directly to the larger memory space ignoring mapping; the screen is always **16K** Bank **5**, no matter where in memory it is (or if it is switched in at all). Setting **D3** of Memory Paging Control (**7FFDh**) will have the ULA read instead from **16K** Bank **7** (otherwise known as "shadow screen"), which can be used as an alternate screen. Beware that this *does not map* **16K** bank **7** into RAM; to alter **16K** bank **7** it must be mapped by other means.

Let's now examine the bit layout of port **1FFDh** used by the +3.

| Bit | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| Description | | | | Par. Port Strobe[6] | Disk Motor[3] | Switch type | ROM / RAM switching | |

When **D0** is **0**, **D1** has no effect and **D2** is a "vertical" ROM switch (ie between **ROM0** and **ROM2** or between **ROM1** and **ROM3**). **D4** at **7FFDh** on the other hand is a "horizontal" ROM switch (ie. between **ROM0** and **ROM1**, or between **ROM2** and **ROM3**). The following diagram illustrates the various ROM switching possibilities:

It is best to think of **D4** in port **7FFDh** and **D2** in port **1FFDh** combining to form a 2-bit number (ranging from **0** to **3**) which determines which ROM occupies the memory area **0000h** to **3FFFh** (**16K** Slot **1**). **D4** of port **7FFDh** is the least significant bit and **D2** of **1FFDh** is the most significant bit.

| D2/1FFDh | D4/7FFDh | ROM Used |
|---|---|---|
| 0 | 0 | **0** |
| 0 | 1 | **1** |
| 1 | 0 | **2** |
| 1 | 1 | **3** |

*ROM switching (with **D0** of **1FFDh** set to **0**)*

Tying it all together, we can easily surmise that 128 style memory management can only alter the bank addressed at **C000h** (For **16K** banks that would be Slot **4**, or for **8K** MMU-type banks Slots **7** and **8**). The active **16K** bank at **C000h** is selected by writing the **3** LSBs of the **16K** bank number to the *bottom 3 bits* of Memory Paging Control (**7FFDh**), and the **4** MSBs to the *bottom 4 bits* of Next Memory Bank Select (**DFFDh**). (The reason for the division is that the original Spectrum 128, having only 128k of memory, only needed 3 bits.)

This in essence constructs a "super hardware port" of sorts, very similar to the combination used to select a ROM using bits from **1FFDh** and **7FFDh**

| D3/DFFDh | D2/DFFDh | D1/DFFDh | D0/DFFDh | D2/7FFDh | D1/7FFDh | D0/7FFDh | Bank |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | **0** |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | **1** |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | **2** |
| ... | | | | | | | |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | **127** |

*"Standard Next paging" bank selection settings*

If you are using the standard interrupt handler or *NextZXOS* routines, then any time you write to the Memory Paging Control port (**7FFDh**) you should also store the value in SysVars at location **5B5Ch**. Any time you write to the +3 Memory Paging Control (**1FFDh**) you should also store the value at **5B67h**. There is no corresponding system variable for the Next-only Next Memory Bank Select (**DFFDh**) port.

6 *Not applicable on the ZX Spectrum Next*

Note that internally *NextZXOS* and *NextBASIC* utilise a combination of all possible banking methods according to what's needed at which time, and you should not rely on this information as a definitive guide on how the system behaves at all times.

### allRAM mode

A "Special paging mode" (also called **allRAM mode or CP/M mode**) is enabled by writing a value with the LSB set to the +3 Memory Paging Control (**1FFDh**). Depending on the 3 low bits of this value a memory configuration is selected as follows:

| D2/1FFDh | D1/1FFDh | D0/1FFDh | RAM Page combinations (Slot1/.../Slot4) |
|---|---|---|---|
| 0 | 0 | 1 | **0, 1, 2, 3** |
| 0 | 1 | 1 | **4, 5, 6, 7** |
| 1 | 0 | 1 | **4, 5, 6, 3** |
| 1 | 1 | 1 | **4, 7, 6, 3** |

*allRAM paging*

This mode is selected by default when you select the *CP/M Menu* from the *More…* submenu of the *Startup menu*, or you run the dot command **.cpm**.

## MMU-Based Memory Management

MMU Based memory management is much simpler to use. It only requires a write to the appropriate MMU Next Register to change the 8K bank occupying a specific 8K slot in the 64K address space (See the previous chapter for details on Next Registers). The MMU registers begin with slot **0** in **NextREG 80** (**50h**) and end with slot **7** in **NextREG 87** (**57h**). For MMU0 and MMU1 only, the ROM can be paged in by selecting **255** (**FFh**) as a bank number. The default values for the MMU Registers are listed in *Chapter 23* and correspond to the normal default memory mapping of the 128K Spectrums.

## Layer 2 Bank Switching

*Layer 2* can also be overlaid on top of the MMU memory map in the bottom 16K or 48K in a Read-only or Write-only mapping. The Write-only mapping, for example, would mean that memory writes to the bottom 16K go to *Layer 2* but memory reads come from the MMU mapping as normal. The bottom 16K is normally occupied by the ROM so this Write-only mapping would allow *NextBASIC* programs to continue to function (the ROM is a read-only program) while allowing **POKE**s to write into the *Layer 2* screen. It is an easy way to gain access to 32K in a single 16K address range.

The *Layer 2* mapping is controlled by bits in the *Layer 2 Access Port* **4667** (**123Bh**). These bits select among 16K or 48K mapping, Read-only or Write-only, and whether the active *Layer 2* screen is mapped or a second *Layer 2* buffer (Shadow Screen) is mapped. *Layer 2* and its second buffer can be located anywhere in RAM and their starting 16K banks are programmed into **NextREG 18** (**12h**) and **19** (**13h**) respectively.

The *Layer 2* mapping does not have to be used for *Layer 2* graphics only; it can be used as a third banking mechanism to access memory more generally.

## Paging method interactions

The most recent change to the memory map, whether that is by Standard or MMU methods, always applies. Each time a change is made to the memory map using the Standard mechanism (a write to port **7FFDh**, **DFFDh**, or **1FFDh**), the affected MMUs are changed immediately. For example writing to port **7FFDh** will change MMU0 and MMU1 to **FFh** to make sure the selected ROM is visible and MMU6 and MMU7 will be changed to reflect the selected 16K RAM bank.

## Paging out the ROM

As seen above, the ROM can be paged out by enabling **allRAM** mode, or by using MMU based memory management. This may cause problems as some programs may assume

that ROM-based service routines are present at fixed addresses in ROM. Additionally, if the default interrupt mode (**IM1**) is set, the CPU will **JP** to **0038h** every frame trying to find an interrupt handler routine. If it does not, (which it won't unless you write your own), the system will crash.

# Chapter 24 – The System Variables

## Overview

Certain locations in memory are set aside for specific uses by the system. There are a few routines (used to keep the paging in order), and some locations called system variables (or SYSVARS). You can use PEEK and DPEEK to read them, in order to find out various things about the system, and on some of them you can usefully change with POKE and DPOKE. They are listed here with their uses.

The area occupied by SYSVARS spans the addresses 23296 (5B00h) to 23733 (5CB5h) (inclusive) and a subset of them are used in 48K BASIC – addresses 23552 (5C00h) to 23733 (5CB5h).

Note that in 48K mode, there is a buffer area between 23296 (5B00h) and 23552 (5C00h) which was used for controlling the printer. This was quite a popular location for small machine code programs on the old 48K Spectrum and if any of these routines are tried in *NextBASIC*, the computer will invariably crash. It's advisable that any 48K BASIC program that uses PEEK, POKE and USR to either be run in 48K BASIC mode (although it can be entered in *NextBASIC* mode and transferred using the SPECTRUM command) or examined thoroughly and converted so it won't use any of these commands. Alternatively, any machine code routine embedded within it should be moved to a safer area.

## System Variables

The system variables listed below, all have unique names, but do not confuse them with *NextBASIC* variables. The computer will not recognize the names as referring to system variables, and they are given solely as mnemonics to be human-readable.

The abbreviations in column 1 of the table that follows have the following meanings:

X The variables should not be poked because the system might crash.
N Poking the variable will have no lasting effect
R Routine entry point. Not a variable.

The number in column 1 is the number of bytes in the variable. For two bytes, the first one is the least significant byte; the reverse of what you might expect. So to POKE a value *v* to a two byte variable at address *n*, use DPOKE instead, as it does the conversion for you. Otherwise you'll need to enter the following for SYSVAR *n* value *v*:

```
POKE n,v-256*INT (v/256)
POKE n+1,INT (v/256)
```

and to peek its value, either use DPEEK or the expression:

```
PEEK n+256*PEEK (n+1)
```

| | Address | | | |
|---|---|---|---|---|
| Notes | Hex | Dec | Name | Description |
| R16 | 5B00 | 23296 | SWAP | Paging subroutine |
| R17 | 5B10 | 23312 | STOO | Paging subroutine. Entered with interrupts already disabled and AF, BC on the stack. |
| R9 | 5B21 | 23329 | YOUNGER | Paging subroutine |
| R16 | 5B2A | 23338 | REGNUOY | Paging subroutine |
| R24 | 5B3A | 23354 | ONERR | Paging subroutine |
| X2 | 5B52 | 23378 | OLDHL | Temporary register store while switching ROMs |
| X2 | 5B54 | 23380 | OLDBC | Temporary register store while switching ROMs |
| X2 | 5B56 | 23382 | OLDAF | Temporary register store while switching ROMs |
| X1 | 5B58 | 23384 | CACHEBNK | 8K Bank ID holding cached program data |
| N1 | 5859 | 23385 | | Reserved for system use |

| Notes | Address Hex | Dec | Name | Description |
|---|---|---|---|---|
| X2 | 5B5A | 23386 | RETADDR | Return address in ROM 1 |
| X1 | 5B5C | 23388 | BANKM | Copy of last byte output to I/O port 7FFDh (32765). See Chapter 23 – The Memory. This byte must be kept up to date with the last value output to the port if interrupts are enabled |
| X1 | 5B5D | 23389 | RAMRST | RST 8 instruction. Used by ROM 1 to report old errors to ROM 3 |
| N1 | 5B5E | 23390 | RAMERR | Error number passed from ROM 1 to ROM 3. Also used by SAVE/LOAD as temporary drive store |
| 1 | 5B5F | 23391 | INKL | INK colour for LoRes |
| 1 | 5B60 | 23392 | INK2 | INK colour for Layer 2 |
| 1 | 5B61 | 23393 | ATTRULA | Attributes for standard mode |
| 1 | 5B62 | 23394 | ATTRHR | Attributes for HiRes (only paper colour in bits 5:3 is used) |
| 1 | 5B63 | 23395 | ATTRHC | Attributes for HiColour |
| 1 | 5B64 | 23396 | INKMASK | Softcopy of EnhancedULA InkMask (or 0) |
| N1 | 5B65 | 23397 | LSBANK | Temporary bank number in LOAD/SAVE and other operations |
| 1 | 5B66 | 23398 | FLAGS3 | Various flags. Bits 0, 1, 6 and 7 unlikely to be useful. Bit 2 is set when tokens are to be expanded on printing. Bit 3 is set if print output is RS232. The default (at reset) is Centronics. Bit 4 is set if a disk interface is present. Bit 5 is set if drive B: is present |
| X1 | 5B67 | 23399 | BANK678 | Copy of last byte output to I/O port 1FFDh (8189). This port is used to control the +3 extended RAM and ROM switching (bits 0..2 – if bit 0 is 0 then bit 2 controls the 'vertical' ROM switch 02 and 13), the disk motor (bit 3) and Centronics strobe (bit 4). This byte must be kept up to date with the last value output to the port if interrupts are enabled |
| X1 | 5B68 | 23400 | FLAGN | Flags for the NextZXOS system |
| 1 | 5B69 | 23401 | MAXBNK | Maximum available RAM bank |
| X2 | 5B6A | 23402 | OLDSP | Old SP (stack pointer) when TSTACK is in use |
| X2 | 5B6C | 23404 | SYNRET | Return address for ONERR |
| 5 | 5B6E | 23406 | LASTV | Last value printed by calculator |
| 1 | 5B73 | 23411 | TILEBNKL | Tiles bank for LoRes |
| 1 | 5B74 | 23412 | TILEML | Tilemap bank for LoRes |
| 1 | 5B75 | 23413 | TILEBNK2 | Tiles bank for Layer2 |
| 1 | 5B76 | 23414 | TILEM2 | Tilemap bank for Layer2 |
| X1 | 5B77 | 23415 | NXTBNK | Bank containing NXTLIN |
| X1 | 5B78 | 23416 | DATABNK | Bank containing DATADD |
| 1 | 5B79 | 23417 | LODDRV | Holds 'T' if LOAD, VERIFY, MERGE are from tape, otherwise holds 'A', 'B' or 'M' |
| 1 | 5B7A | 23418 | SAVDRV | Holds 'T' if SAVE is to tape, otherwise holds 'A', 'B' or 'M' |
| N1 | 5B7B | 23419 | L2SOFT | Softcopy of Layer 2 port |
| 2 | 5B7C | 23420 | TILEWL | Width of LoRes tilemap |
| 2 | 5B7E | 23422 | TILEW2 | Width of Layer 2 tilemap |
| 2 | 5B80 | 23424 | TILEOFFL | Offset in bank for LoRes tilemap |
| 2 | 5B82 | 23426 | TILEOFF2 | Offset in bank for Layer 2 tilemap |
| 2 | 5B84 | 23428 | COORDSX | X Coordinate of last point plotted (Layer 1/2) |
| 2 | 5B86 | 23430 | COORDSY | Y Coordinate of last point plotted (Layer 1/2) |
| 1 | 5B88 | 23432 | PAPERL | PAPER colour for LoRes mode |
| 1 | 5B89 | 23433 | PAPER2 | PAPER colour for Layer 2 mode |
| Nx | 5B8A | 23434 | TMPVARS | Base of temporary system variables (space shared with bottom of TSTACK) |

| Notes | Address Hex | Dec | Name | Description |
|---|---|---|---|---|
| X117 | 5BFF | 23551 | TSTACK | Temporary stack grows down from here. Used when RAM bank 7 is switched in at top of memory while executing the editor or calling NextZXOS. It may safely go down to 5B8Ah if necessary. This guarantees at least 117 bytes of stack when NextBASIC calls NextZXOS |
| N8 | 5C00 | 23552 | KSTATE | Used in reading the keyboard |
| N1 | 5C08 | 23560 | LASTK | Stores newly pressed key |
| 1 | 5C09 | 23561 | REPDEL | Time (in 50ths of a second) that a key must be held down before it repeats. This starts off at 35, but you can POKE in other values |
| 1 | 5C0A | 23562 | REPPER | Delay (in 50ths of a second) between successive repeats of a key held down – initially 5) |
| X2 | 5C0B | 23563 | RETVARS | Address of local variables on return stack |
| N1 | 5C0D | 23565 | K_DATA | Stores 2nd byte of colour controls entered from keyboard |
| N2 | 5C0E | 23566 | TVDATA | Stores bytes of colour, AT and TAB controls going to TV |
| X38 | 5C10 | 23568 | STRMS | Addresses of channels attached to streams |
| 2 | 5C36 | 23606 | CHARS | 256 less than address of character set (which starts with space and carries on until ©). Normally in ROM, but you can set up your own in RAM and make CHARS point to it |
| 1 | 5C38 | 23608 | RASP | Length of warning buzz |
| 1 | 5C39 | 23609 | PIP | Length of keyboard click |
| 1 | 5C3A | 23610 | ERRNR | 1 less than the report code. Starts off at 255 (for -1) so PEEK 23610 gives 255 |
| X1 | 5C3B | 23611 | FLAGS | Various flags to control the NextBASIC system |
| X1 | 5C3C | 23612 | TVFLAG | Flags associated with the TV |
| X2 | 5C3D | 23613 | ERRSP | Address of item on machine stack to be used as error return |
| N2 | 5C3F | 23615 | | Reserved for system use |
| N1 | 5C41 | 23617 | MODE | Specifies K, L, C, E or G cursor. |
| 2 | 5C42 | 23618 | NEWPPC | Line to be jumped to |
| N1 | 5C44 | 23620 | | Reserved for system use |
| 2 | 5C45 | 23621 | PPC | Line number of statement currently being executed |
| 1 | 5C47 | 23623 | SUBPPC | Number within line of statement currently being executed |
| 1 | 5C48 | 23624 | BORDCR | Border colour multiplied by 8; also contains the attributes normally used for the lower half of the screen |
| 2 | 5C49 | 23625 | E_PPC | Number of current line (with program cursor) |
| X2 | 5C4B | 23627 | VARS | Address of variables |
| N2 | 5C4D | 23629 | DEST | Address of variable in assignment |
| X2 | 5C4F | 23631 | CHANS | Address of channel data |
| X2 | 5C51 | 23633 | CURCHL | Address of information currently being used for input and output |
| X2 | 5C53 | 23635 | PROG | Address of NextBASIC program |
| X2 | 5C55 | 23637 | NXTLIN | Address of next line in program |
| X2 | 5C57 | 23639 | DATADD | Address of terminator of last DATA item |
| X2 | 5C59 | 23641 | E_LINE | Address of command being typed in |
| 2 | 5C5B | 23643 | K_CUR | Address of cursor |
| X2 | 5C5D | 23645 | CH_ADD | Address of the next character to be interpreted – the character after the argument of PEEK, or the NEWLINE at the end of a POKE statement |
| 2 | 5C5F | 23647 | X_PTR | Address of the character after the ? marker |
| X2 | 5C61 | 23649 | WORKSP | Address of temporary work space |
| X2 | 5C63 | 23651 | STKBOT | Address of bottom of calculator stack |
| X2 | 5C65 | 23653 | STKEND | Address of start of spare space |
| N1 | 5C67 | 23655 | BREG | Calculator's B register |

| Notes | Address Hex | Dec | Name | Description |
|---|---|---|---|---|
| N2 | 5C68 | 23656 | MEM | Address of area used for calculator's memory (usually MEMBOT, but not always) |
| 1 | 5C6A | 23658 | FLAGS2 | More flags. (Bit 3 set when CAPS SHIFT or CAPS LOCK is on) |
| X1 | 5C6B | 23659 | DF_SZ | The number of lines (including one blank line) in the lower part of the screen |
| N2 | 5C6C | 23660 | | Reserved for system use |
| 2 | 5C6E | 23662 | OLDPPC | Line number to which CONTINUE jumps |
| 1 | 5C70 | 23664 | OSPPC | Number within line of statement to which CONTINUE jumps |
| N1 | 5C71 | 23665 | FLAGX | Various flags |
| N2 | 5C72 | 23666 | STRLEN | Length of string type destination in assignment |
| N2 | 5C74 | 23668 | T_ADDR | Address of next item in syntax table (very unlikely to be useful) |
| 2 | 5C76 | 23670 | SEED | The seed for RND. This is the variable that is set by RANDOMIZE |
| 3 | 5C78 | 23672 | FRAMES | 3 byte (least significant byte first), frame counter incremented every 20ms |
| 2 | 5C7B | 23675 | UDG | Address of first user-defined graphic. You can change this, for instance, to save space by having fewer user-defined characters |
| 1 | 5C7D | 23677 | COORDS | X-coordinate of last point plotted |
| 1 | 5C7E | 23678 | | Y-coordinate of last point plotted |
| X1 | 5C7F | 23679 | GMODE | Graphical layer/mode flags |
| X1 | 5C80 | 23680 | PRCC | Full address of next position for LPRINT to print at (in ZX Printer buffer). Legal values 5B00 – 5B1F[1] |
| 1 | 5C81 | 23681 | STIMEOUT | Screensaver control |
| 2 | 5C82 | 23682 | ECHO_E | 33-column number and 24 line number (in lower half) of end of input buffer |
| 2 | 5C84 | 23684 | DF_CC | Address in display file of PRINT position |
| 2 | 5C86 | 23686 | DF_CCL | Like DF_CC for lower part of screen |
| X1 | 5C88 | 23688 | S_POSN | 33-column number for PRINT position |
| X1 | 5C89 | 23689 | | 24-line number for PRINT position |
| X2 | 5C8A | 23690 | SPOSNL | Like S_POSN for lower part |
| 1 | 5C8C | 23692 | SCR_CT | Counts scrolls – it is always 1 more than the number of scrolls that will be done before stopping with scroll?. If you keep poking this with a number bigger than 1 (say 255), the screen will scroll on and on without asking you |
| 1 | 5C8D | 23693 | ATTR_P | Permanent current colours, etc., (as set up by colour statements) |
| 1 | 5C8E | 23694 | MASK_P | Used for transparent colours, etc. Any bit that is 1 shows that the corresponding attribute bit is taken not from ATTR_P, but from what is already on the screen |
| N1 | 5C8F | 23695 | ATTR_T | Temporary current colours, etc., (as set up by colour items) |
| N1 | 5C90 | 23696 | MASK_T | Like MASK_P, but temporary |
| 1 | 5C91 | 23697 | P_FLAG | More flags |
| N30 | 5C92 | 23698 | MEMBOT | Calculator's memory area – used to store numbers that cannot conveniently be put on the calculator stack |
| 2 | 5CB0 | 23728 | | Not used |
| 2 | 5CB2 | 23730 | RAMTOP | Address of last byte of NextBASIC system area. |
| 2 | 5CB4 | 23732 | P_RAMT | Address of last byte of physical RAM. |

1 *Not used in 128K mode or when certain peripherals are attached*

# Chapter 25 – Using Machine Code

## Using Machine Code

Computers do not respond directly to BASIC, or any other higher level programming language. Instead, such languages are either interpreted or compiled into what is known as *machine code*, and it is this which is understood by the CPU. The kind of processor that is built into the computer determines the type of machine code that is used. The ZX Spectrum range contains a Z80 processor, and so one writes Z80 machine code when addressing the processor directly. Specifically for the ZX Spectrum Next, the CPU is an updated one called Z80N which contains a superset of the instructions found in the Z80. This section is written mainly for those who understand Z80 machine code. If you do not, but would like to, you might choose to read a book about it. Suitable titles will be something along the lines of *Z80 machine code (or assembly language) for the absolute beginner*. If it also mentions one of the computers in the ZX Spectrum range, so much the better. You might also like to read online resources and find tools such as the Design-Design **Zeus** *cross-assembler*[1] at: **https://www.desdes.com/products/oldfiles/**, or the **z88dk** suite which includes apart from a C compiler, also an assembler, at **www.z88dk.org**, and last but not least the **specnext.com** forums.

Rather than write the numerical values of a machine code program directly, people usually choose to use mnemonics, known as assembly language, which, although cryptic, is not too difficult to understand with practice. You can see the assembly language instructions understood by the ZX Spectrum Next's CPU in *Appendix A*.

For a computer to execute this code the program must be converted into a sequence of bytes – in this form it is called machine code. This translation is usually done by a computer, using a program called an assembler. There is no assembler built into the ZX Spectrum Next ROM, however, three, loadable ones are included in the **System/Next™** distribution: **Zeus, Odin** and **SPED** kindly provided by Neil Mottershead and Simon Brattel for the first, Matt Davies for the second and César Hernández Bañó for the latter respectively. It is also possible do the translation yourself, but this can be a painstaking process.

Let's take as an example the program:

```
ld bc, 99
ret
```

This will load the **BC** register pair with **99** and then return. This translates into the four machine code bytes **1**, **99, 0** (**ld bc, 99**) and **201** (**ret**). (If you look up codes **1** and **201** in *Appendix A*, you will find that **1** corresponds to **ld bc, NN** – where **NN** stands for any two-byte number; and **201** corresponds to **ret**.)

## Using CLEAR to Make Space

Once you have written your machine code program, the next step is to load it into the computer's memory. You need to decide whereabouts in memory to locate it – the best thing is to make extra space for it between the *NextBASIC* area and the user-defined graphics.

If you enter the command:

```
CLEAR 65267
```

This will give you a space of **100** (for good measure) bytes starting at address **65268**.To create the machine code program, you may run a *NextBASIC* program like this:

```
10 a=65268
20 READ n: POKE a,n
```

1 *A cross-assembler is an assembler that runs on a different system than the one it produces code for. For example z88dk is usually executed on a PC running Linux or Windows Operating Systems.*

```
30 a+=1: GO TO 20
40 DATA 1,99,0,201
```

This will stop with the report E Out of DATA when it has filled in the four bytes you specified.

## Using USR to run machine code

To run the machine code, you use the function USR or its –preferred– BANK command variant. In its simplest form USR must be provided with a numeric argument, i.e. the starting address or the bank offset. Its result is the value of the BC register on return from the machine code program, so assuming you type:

```
PRINT USR 65268
```

It will return the value 99.

The return address to *NextBASIC* is stacked in the usual way, so return is by a Z80 ret instruction. You should not use the IY and I registers in a machine code routine that expects to use the *NextBASIC* interrupt mechanism. To perform the exact same function by using the BANK variant, make the following changes to our program:

```
10 %a=0
20 BANK NEW %b
30 READ %n : BANK %b POKE %a,%n
40 %a+=1: GO TO 30
50 DATA 1,99,0,201
```

RUN it and you'll see the E Out of Data error again; Now it's time to execute it and it's done by giving:

```
PRINT % BANK b USR 0
```

There are a few more variants of USR that differ in key points and make the life of the machine-code programmer a bit easier. These are:

USR$ *addr*
BANK *n* USR$ *offset*

which call the machine code routine at *addr* (or *offset* in bank *n*). Instead however of returning the 16-bit number found in BC (as with USR *addr*), USR$ returns a string, defined by the start address returned by the machine-code routine in DE and length in BC.

Additionally, USR as well as USR$ (and their BANK variants) can be provided with optional parameters like:

USR(*addr, param1*[, *param2* [, *param3...*]]])

USR$(*addr, param1*[, *param2* [, *param3...*]]])

BANK *n* USR(*addr, param1*[, *param2* [, *param3...*]]])

BANK *n* USR$(*addr, param1*[, *param2* [, *param3...*]]])

which can be passed to the machine-code routine (instead of just the start address in BC).

If a single additional parameter (*param1*) is present, this is passed in BC (if it is numeric) or as an address DE and length BC (if it is a string).

The type of the parameter passed to the routine is indicated by the *zero flag*: if set, the parameter is a string (in DE,BC); if clear, the parameter is a number in BC.

Additionally, the type of the expected result is indicated by the carry flag: if set, the expected result is a number (in **BC**); if clear, the expected result is a string (in **DE**,**BC**).

All further parameters are left on the calculator stack for the machine-code routine to use with calculator operations or retrieve using standard ROM routines such as FIND-INT1, FIND-INT2, STK-FETCH. On entry, **A** contains the number of additional parameters on the calculator stack (**0-16**) and **HL** contains a bitmask indicating the type of each parameter. Bit **15**, indicates the type of the final parameter (and will be the first to be retrieved from the calculator stack), so types can be read by shifting each bit in turn to the carry flag with **ADD HL,HL**. Type bits are **0** for string, **1** for numeric.

Routines must remove all additional parameters from the calculator stack, otherwise it will be unbalanced and the expression may be calculated incorrectly.

If you are writing a program to run with the 48K or 128K ROM, you should not load **I** with values between **40h** and **7Fh** (even if you never use IM 2). When using one of the 128K ROMs, values between **C0h** and **FFh** for **I** should also be avoided if you plan on enabling contention for your target machine / personality and contended memory (i.e. RAM 4 to 7) is to be paged in between **C000h** and **FFFFh**. This is due to an interaction between the ULA and the Z80 refresh mechanism, which can cause apparently inexplicable crashes, screen corruption or other undesirable effects. Thus, you should only use vector IM 2 interrupts between **8000h** and **BFFFh** unless you are very confident of your memory mapping (or you are only going to run your program on the +2A, +3e or Next personalities where this problem does not exist).

There are a number of standard pitfalls when programming a banked system such as the ZX Spectrum Next from machine code. If you are experiencing problems, check that your stack is not being paged out during interrupts, and that your interrupt routine is always where you expect it to be (it is advisable to disable interrupts during paging operations). It is also recommended that you keep a copy of the current bank register setting in unpaged RAM somewhere as the ports are write-only. *NextBASIC* and the editor use the system variables BANKM and BANK678 for **7FFDh** and **1FFDh** respectively.

If you call *NextZXOS* routines, remember that interrupts should be enabled upon entry to the routines. Remember also that the stack must be below **49120** (**BFE0h**) and above **16384** (**4000h**), and that there must be at least **50** words of stack space available.

You can save your machine code program easily enough with, for example:

```
SAVE "name" CODE 65268,4
```

or, in case you used the **BANK** variant

```
SAVE "name" BANK %b, 0, 4
```

There is no way of saving the program such that when loaded it automatically runs itself; however, you can get round this by using the short *NextBASIC* program:

```
10 LOAD "name" CODE 65268,4
20 PRINT USR 65268
```

Which should also be saved as a separate program, using a command of the following form:

```
SAVE "loader" LINE 10
```

You may run the machine code from *NextBASIC* using the single command:

```
LOAD "loader"
```

This then loads and automatically runs the *NextBASIC* program, which in turn loads and runs the machine code. You can try and make a version with the **BANK** variant as well as that's safer and always preferred.

## Calling NextZXOS from NextBASIC

When *NextBASIC*'s **USR** function is used, the code it references is entered with the memory configured with the ROM switched in at the bottom of memory in the address range (000h – 3FFFh) being ROM 3 (the 48 BASIC ROM). The RAM page at the top of memory is Bank 0 and the machine stack resides in this area (unless the **CLEAR** command has been used to reduce it to somewhere below **C000h**). As explained in the accompanying documents explaining the *NextZXOS* API (found in the c:/docs/nextzxos/ folder in your System/Next™ distribution), *NextZXOS* can only be called with RAM page 7 switched in at the top of memory, the stack held somewhere in that range **4000h** to **BFE0h**, and ROM 2 (the *NextZXOS* ROM) switched in at the bottom of memory (**000h** to **3FFFh**).

Consequently, it will be necessary to switch both ROM and RAM, and move the stack before and after calling one of the entries in the DOS jump table.

If the **CLEAR** command has been used so that the *NextBASIC stack* is below **49120** (**BFE0h**), then it is not necessary to move the stack. However, we have done so in the following example to demonstrate the technique when this is not the case.

A simple example to call DOS CATALOG:

```
                org  7000h

mystak          equ  9FFFh              ;arbitrary value picked to be below
                                        ;BFE0h and above 4000h
staksto         equ  9000h              ;somewhere to put BASIC's stack
                                        ;pointer
bankm           equ  5B5Ch              ;system variable that holds the
                                        ;last value output to 7FFDh
port1           equ  7FFDh              ;address of ROM/RAM switching port
                                        ;in I/O map
catbuff         equ  8000h              ;somewhere for DOS to put its cata
                                        ;log
dos_catalog     equ  011Eh              ;the DOS routine to call
demo:           di                      ;unwise to switch RAM/ROM without
                                        ;disabling interrupts
                ld   (staksto),sp       ;save BASIC's stack pointer
                ld   bc,port1           ;the horizontal ROM switch/RAM
                     ;switch I/O address
                ld   a,(bankm)          ;system variable that holds current
                                        ;switch state
                res  4,a                ;move right to left in horizontal
                                        ;ROM switch (3 to 2)
                or   7                  ;switch in RAM page 7
                ld   (bankm),a          ;must keep system variable up to
                                        ;date (very important)
                out  (c),a              ;make the switch
                ld sp,mystak            ;make sure stack is above 4000h and
                                        ;below BFE0h
                ei                      ;interrupts can now be enabled
                                        ;
                                        ;The above will have switched in
                                        ;the DOS ROM and RAM page 7. The
                                        ;stack has also been located in a
                                        ;"safe" position for calling DOS
                                        ;
                                        ;The following is the code to set
                                        ;up and call DOS CATALOG. This is
                                        ;where yourown code would be
                                        ;placed.
                                        ;
                ld hl,catbuff           ;somewhere for DOS to put the cata
                                        ;log
                ld de,catbuff+1         ;
```

```
                ld bc,1024              ;maximum (for +3DOS) is actually
                                        ;64x13+13 = 845
                ld (hl),0
                ldir                    ;make sure at least first entry is
                                        ;zeroed
                ld b,64                 ;the number of entries in the
                                        ;buffer
                ld c,1                  ;include system files in the cata
                                        ;log
                ld de,catbuff           ;the location to be filled with the
                                        ;disk catalog
                ld hl,stardstar         ;the file name ("*.*")
                call dos_catalog        ;call the DOS entry
                push af                 ;save flags and possible error num
                                        ;ber returned by DOS
                pop hl
                ld (dosret),hl          ;put it where it can be seen from
                                        ; NextBASIC
                ld c,b                  ;move number of files in catalog to
                                        ;low byte of BC
                ld b,0                  ;this will be returned in NextBASIC
                                        ;by the USR function
                                        ;
                                        ;If the above worked, then BC holds
                                        ;number of files in catalog, the
                                        ;"catbuff"
                                        ;will be filled with the alpha-
                                        ;numerically sorted catalog and the
                                        ;carry flag but
                                        ;in "dosret" will be set. This will
                                        ;be peeked from NextBASIC to check
                                        ;if all went well.
                                        ;
                                        ;Having made the call to DOS, it is
                                        ;now necessary to undo the ROM and
                                        ;RAM switch and put BASIC's stack
                                        ;back to where it was on entry.
                                        ;The following will achieve this.
                di                      ;about to ROM/RAM switch so be
                                        ;careful
                push bc                 ;save number of files
                ld   bc,port1           ;I/O address of horizontal ROM/RAM
                                        ;switch
                ld   a,(bankm)          ;get current switch state
                set  4,a                ;move left to right (ROM 2 to ROM
                                        ;3)
                and  F8h                ;also want RAM page 0
                ld   (bankm),a          ;update the system variable (very
                     ;important)
                out  (c),a              ;make the switch
                pop  bc                 ;get back the saved number of files
                                        ;in catalog
                ld   sp,(staksto)       ;put NextBASIC's stack back
                ret                     ;return to NextBASIC, value in BC
                                        ;is returned to USR
stardstar:
                defb "*.*",FFh          ;the file name, must be terminated
                     ;with FFh
dosret:
                defw 0                  ;a variable to be peeked from BASIC
                                        ;to see if it worked
```

As some of you may not have an assembler available, the following is a *NextBASIC* program that pokes the above code into memory, calls it, and then uses the value returned by the **USR** function and the contents of **dosret** to print a very simple catalog of the disk:

```
 10 sum=0
 20 FOR i=28672 TO 28758
 30    READ n
 40    POKE i,n : sum+=n
 50 NEXT i
 60 IF sum <> 9387 THEN PRINT
    "Error in DATA" : STOP
 70 x= USR 28672
 80 IF INT ( PEEK (28757)/2)=
    PEEK (28757)/2 THEN PRINT
    "Disk Error ";PEEK
    (28758): STOP
 90 IF x=1 THEN PRINT "No file
    found": STOP
100 FOR i=0 TO x-2
110 FOR j=0 TO 10
120 PRINT CHR$ ( PEEK
    (32781+i*13+j));
130 NEXT j
140 PRINT
150 NEXT i
160 DATA 243,237,115,0,144,1,
    253,127,58,92,91,203,167,2
    46,7,50,92,91,237,121,49,2
    55,159,251
170 DATA 33,0,128,17,1,128,1,
    0,4,54,0,237,176,6,64,14,1
    ,17,0,128,33,81,112,205,30
    ,1,245,225,34,85,112,72,6,
    0
180 DATA 243,197,1,253,127,58,
    92,91,203,231,230,248,50,9
    2,91,237,121,193,237,123,0
    , 144,201
190 DATA 42,46,42,255,0,0
```

The addresses picked for the above code and its data areas are completely arbitrary. However, it is a good idea to keep things in the central 32K wherever possible so as not to run into the pitfall of accidentally switching out a vital variable or piece of code.

If interrupts are to be enabled (as is the case in the above example), it is imperative that the system is kept up to date about the latest ROM switch. This means, that the user must make the BANK678 system variable reflect the last value output to the port at 1FFDh. As shown by the above example, the general technique is to take a copy of the variable in A, set/reset the relevant bits, update the system variable then make the switch with an OUT instruction. Interrupts must be disabled while the system variable does not reflect the cur-

rent state of the port. The port at 1FFDh doesn't just control the ROM switch, so setting the variable to absolute values would be very unwise. Using AND/OR with a bit mask or SET/RES instructions is the preferred method of updating the variable.

Just as BANK678 reflects the last value output to 1FFDh, BANKM should also be kept up to date with the last value output to 7FFDh. Again, it is unwise to use absolute values, as the port is used for other purposes. For example, the bottom 3 bits of the port are used to select the RAM page that is switched into the memory area C000h through FFFFh (this is also shown in the above example). Naturally, when more than one bit is to be set/reset, a bit mask used with OR/AND is the more efficient method. Note that RAM paging was described in the *Memory Management section* in *Chapter 24*.

The above was a very simple example of calling DOS routines. It works – apart from the ZX Spectrum Next – on the ZX Spectrum +3 and ZX Spectrum +3e as well.

## Opcodes Prefixes

Some Assembler opcodes are preceded by a prefix byte which changes the opcode represented by the following byte.

Assembler opcode prefixes CBh (203) and EDh (237) alter the meaning of certain instructions, as indicated in the 5th and 6th columns of *Appendix A*. This includes the provision of some entirely new opcodes for the ZX Spectrum Next.

Assembler opcode prefixes DDh (221) and FDh (253) alter the meaning of certain instructions that ordinarily refer to the H or L registers, so that they refer to either the component registers of IX or IY register respectively. For example, the instruction LD H,n will load the value of n into the H register. Preceding this two-byte instruction with the IX register's opcode prefix DDh, would result in the most significant 8 bits of the IX register being loaded with that value instead.

This general transformation rule is modified when the original instruction contains (HL), with this component replaced by (IX +N) and any other reference to HL left unaffected. For instance:

DDh 66h is interpreted as ld h,(ix + N)

A DDh opcode will be ignored, interpreted as nop, if it precedes DDh, EDh or FDh. Similar rules apply to the FDh instruction.

# Appendix A
## Character Set, Z80N Mnemonics and Control Codes

This is the complete ZX Spectrum Next / *NextZXOS* character set, with codes in decimal and hexadecimal, the character each code represents, as well as the control codes (shaded) together with their corresponding *NextBASIC* tokens, if any. Tokens that are shaded are specific to the ZX Spectrum Next and cannot be found in earlier ZX Spectrum models. As the codes are also Z80N machine code instructions, the right hand columns give the corresponding assembly language mnemonics. As you are probably aware if you understand these things, certain Z80N instructions are compounds starting with **CBh** or **EDh**; the two rightmost columns give you these. Note that **ED** instructions that are shaded, cannot be found in regular Z80 CPUs and are only native to Z80N, the variant of the Z80 CPU, found on the ZX Spectrum Next. Control codes are marked with **UW** if they refer to User Windows and **SW** if they refer to System Windows.

| Dec | Character / Control Code / Token | Hex | Z80N Assembler | - after CB | - after ED |
|---|---|---|---|---|---|
| 0 | **Justify off** (UW) **Increase font** (SW) | 00 | nop | rlc b | |
| 1 | **Justify on** (UW) **Decrease font** (SW) | 01 | ld bc,NN | rlc c | |
| 2 | **Save Window** (UW) **Change font** (SW) | 02 | ld (bc),a | rlc d | |
| 3 | **Restore Window** (UW)<br>**Regenerate Small Fonts** (SW) | 03 | inc bc | rlc e | |
| 4 | **Cursor to top left** (UW)(SW) | 04 | inc b | rlc h | |
| 5 | **Cursor to bottom left** (UW)(SW) | 05 | dec b | rlc l | |
| 6 | **PRINT** comma | 06 | ld b,N | rlc (hl) | |
| 7 | **EDIT**, **Scroll** (SW)(UW) | 07 | rlca | rlc a | |
| 8 | ⇦ | 08 | ex af,af' | rrc b | |
| 9 | ⇨ | 09 | add hl,bc | rrc c | |
| 10 | ⇩ | 0A | ld a,(bc) | rrc d | |
| 11 | ⇧ | 0B | dec bc | rrc e | |
| 12 | DELETE / **Backspace** | 0C | inc c | rrc h | |
| 13 | ENTER / Carriage Return<br>**PRINT** apostrophe | 0D | dec c | rrc l | |
| 14 | **Clear Window** (UW)(SW) | 0E | ld c,N | rrc (hl) | |
| 15 | **Wash Window** (UW)(SW) | 0F | rrca | rrc a | |
| 16 | **INK** | 10 | djnz DIS | rl b | |
| 17 | **PAPER** | 11 | ld de,NN | rl c | |
| 18 | **FLASH** | 12 | ld (de),a | rl d | |
| 19 | **BRIGHT** | 13 | inc de | rl e | |
| 20 | **INVERSE** | 14 | inc d | rl h | |
| 21 | **OVER** | 15 | dec d | rl l | |
| 22 | **AT** | 16 | ld d,N | rl (hl) | |
| 23 | **TAB** | 17 | rla | rl a | |
| 24 | **ATTR** (UW)(SW) | 18 | jr DIS | rr b | |
| 25 | **POINT** (UW)(SW) | 19 | add hl,de | rr c | |
| 26 | **AUTO PAUSE** (UW)(SW) | 1A | ld a,(de) | rr d | |
| 27 | **Fill window with character**(UW)(SW) | 1B | dec de | rr e | |
| 28 | **Set Double Width** (UW)(SW) | 1C | inc e | rr h | |
| 29 | **Set Font Height** (UW) | 1D | dec e | rr l | |
| 30 | **Justification mode** (UW)<br>**Set Font Width** (SW) | 1E | ld e,N | rr (hl) | |
| 31 | **Permit embed. codes in justif. mode** (UW)<br>**Redefine Character Set** (SW) | 1F | rra | rr a | |
| 32 | Space | 20 | jr nz, DIS | sla b | |
| 33 | ! | 21 | ld hl,NN | sla c | |

| Dec | Character / Control Code / Token | Hex | Z80N Assembler | - after CB | - after ED |
|---|---|---|---|---|---|
| 34 | " | 22 | ld (NN),hl | sla d | |
| 35 | # | 23 | inc hl | sla e | swapnib |
| 36 | $ | 24 | inc h | sla h | mirror a |
| 37 | % | 25 | dec h | sla l | |
| 38 | & | 26 | ld h,N | sla (hl) | |
| 39 | ‘ | 27 | daa | sla a | test N |
| 40 | ( | 28 | jr z,DIS | sra b | bsla de,b |
| 41 | ) | 29 | add hl,hl | sra c | bsra de,b |
| 42 | * | 2A | ld hl,(NN) | sra d | bsrl de,b |
| 43 | + | 2B | dec hl | sra e | bsrf de,b |
| 44 | , | 2C | inc l | sra h | brlc de,b |
| 45 | - | 2D | dec l | sra l | |
| 46 | . | 2E | ld l,N | sra (hl) | |
| 47 | / | 2F | cpl | sra a | |
| 48 | 0 | 30 | jr nc,DIS | | mul d,e |
| 49 | 1 | 31 | ld sp,NN | | add hl,a |
| 50 | 2 | 32 | ld (NN),a | | add de,a |
| 51 | 3 | 33 | inc sp | | add bc,a |
| 52 | 4 | 34 | inc (hl) | | add hl,NN |
| 53 | 5 | 35 | dec (hl) | | add de,NN |
| 54 | 6 | 36 | ld (hl),N | | add bc,NN |
| 55 | 7 | 37 | scf | | |
| 56 | 8 | 38 | jr c,DIS | srl b | |
| 57 | 9 | 39 | add hl,sp | srl c | |
| 58 | : | 3A | ld a,(NN) | srl d | |
| 59 | ; | 3B | dec sp | srl e | |
| 60 | < | 3C | inc a | srl h | |
| 61 | = | 3D | dec a | srl l | |
| 62 | > | 3E | ld a,N | srl (hl) | |
| 63 | ? | 3F | ccf | srl a | |
| 64 | @ | 40 | ld b,b | bit 0,b | in b,(c) |
| 65 | A | 41 | ld b,c | bit 0,c | out (c),b |
| 66 | B | 42 | ld b,d | bit 0,d | sbc hl,bc |
| 67 | C | 43 | ld b,e | bit 0,e | ld (NN),bc |
| 68 | D | 44 | ld b,h | bit 0,h | neg |
| 69 | E | 45 | ld b,l | bit 0,l | retn |
| 70 | F | 46 | ld b,(hl) | bit 0,(hl) | im 0 |
| 71 | G | 47 | ld b,a | bit 0,a | ld i,a |
| 72 | H | 48 | ld c,b | bit 1,b | in c,(c) |
| 73 | I | 49 | ld c,c | bit 1,c | out (c),c |
| 74 | J | 4A | ld c,d | bit 1,d | adc hl,bc |
| 75 | K | 4B | ld c,e | bit 1,e | ld bc,(NN) |
| 76 | L | 4C | ld c,h | bit 1,h | |
| 77 | M | 4D | ld c,l | bit 1,l | reti |
| 78 | N | 4E | ld c,(hl) | bit 1,(hl) | |
| 79 | O | 4F | ld c,a | bit 1,a | ld r,a |
| 80 | P | 50 | ld d,b | bit 2,b | in d,(c) |
| 81 | Q | 51 | ld d,c | bit 2,c | out (c),d |
| 82 | R | 52 | ld d,d | bit 2,d | sbc hl,de |
| 83 | S | 53 | ld d,e | bit 2,e | ld (NN),de |
| 84 | T | 54 | ld d,h | bit 2,h | |
| 85 | U | 55 | ld d,l | bit 2,l | |

| Dec | Character / Control Code / Token | | Hex | Z80N Assembler | - after CB | - after ED |
|---|---|---|---|---|---|---|
| 86 | V | | 56 | ld d,(hl) | bit 2,(hl) | im 1 |
| 87 | W | | 57 | ld d,a | bit 2,a | ld a,i |
| 88 | X | | 58 | ld e,b | bit 3,b | in e,(c) |
| 89 | Y | | 59 | ld e,c | bit 3,c | out (c),e |
| 90 | Z | | 5A | ld e,d | bit 3,d | adc hl,de |
| 91 | [ | | 5B | ld e,e | bit 3,e | ld de,(NN) |
| 92 | \ | | 5C | ld e,h | bit 3,h | |
| 93 | ] | | 5D | ld e,l | bit 3,l | |
| 94 | ↑ | | 5E | ld e,(hl) | bit 3,(hl) | im 2 |
| 95 | _ | | 5F | ld e,a | bit 3,a | ld a,r |
| 96 | £ | | 60 | ld h,b | bit 4,b | in h,(c) |
| 97 | a | | 61 | ld h,c | bit 4,c | out (c),h |
| 98 | b | | 62 | ld h,d | bit 4,d | sbc hl,hl |
| 99 | c | | 63 | ld h,e | bit 4,e | ld (NN),hl |
| 100 | d | | 64 | ld h,h | bit 4,h | |
| 101 | e | | 65 | ld h,l | bit 4,l | |
| 102 | f | | 66 | ld h,(hl) | bit 4,(hl) | |
| 103 | g | | 67 | ld h,a | bit 4,a | rrd |
| 104 | h | | 68 | ld l,b | bit 5,b | in l,(c) |
| 105 | i | | 69 | ld l,c | bit 5,c | out (c),l |
| 106 | j | | 6A | ld l,d | bit 5,d | adc hl,hl |
| 107 | k | | 6B | ld l,e | bit 5,e | ld hl,(NN) |
| 108 | l | | 6C | ld l,h | bit 5,h | |
| 109 | m | | 6D | ld l,l | bit 5,l | |
| 110 | n | | 6E | ld l,(hl) | bit 5,(hl) | |
| 111 | o | | 6F | ld l,a | bit 5,a | rld |
| 112 | p | | 70 | ld (hl),b | bit 6,b | in f,(c) |
| 113 | q | | 71 | ld (hl),c | bit 6,c | |
| 114 | r | | 72 | ld (hl),d | bit 6,d | sbc hl,sp |
| 115 | s | | 73 | ld (hl),e | bit 6,e | ld (NN),sp |
| 116 | t | | 74 | ld (hl),h | bit 6,h | |
| 117 | u | | 75 | ld (hl),l | bit 6,l | |
| 118 | v | | 76 | halt | bit 6,(hl) | |
| 119 | w | | 77 | ld (hl),a | bit 6,a | |
| 120 | x | | 78 | ld a,b | bit 7,b | in a,(c) |
| 121 | y | | 79 | ld a,c | bit 7,c | out (c),a |
| 122 | z | | 7A | ld a,d | bit 7,d | adc hl,sp |
| 123 | { | | 7B | ld a,e | bit 7,e | ld sp,(NN) |
| 124 | \| | | 7C | ld a,h | bit 7,h | |
| 125 | } | | 7D | ld a,l | bit 7,l | |
| 126 | ~ | | 7E | ld a,(hl) | bit 7,(hl) | |
| 127 | © | | 7F | ld a,a | bit 7,a | |
| 128 | | | 80 | add a,b | res 0,b | |
| 129 | ■ | **TIME** | 81 | add a,c | res 0,c | |
| 130 | ■ | **PRIVATE** | 82 | add a,d | res 0,d | |
| 131 | ■ | **IFELSE**[1] | 83 | add a,e | res 0,e | |
| 132 | ■ | **ENDIF** | 84 | add a,h | res 0,h | |
| 133 | ■ | **EXIT** | 85 | add a,l | res 0,l | |
| 134 | ■ | **REF** | 86 | add a,(hl) | res 0,(hl) | |
| 135 | ■ | **PEEK$** | 87 | add a,a | res 0,a | |

1 *Displays as IF but indicates ELSE*

| Dec | Character / Control Code / Token | Hex | Z80N Assembler | - after CB | - after ED |
|---|---|---|---|---|---|
| 136 | REG | 88 | adc a,b | res 1,b | |
| 137 | DPOKE | 89 | adc a,c | res 1,c | |
| 138 | DPEEK | 8A | adc a,d | res 1,d | push NN |
| 139 | MOD | 8B | adc a,e | res 1,e | |
| 140 | << | 8C | adc a,h | res 1,h | |
| 141 | >> | 8D | adc a,l | res 1,l | |
| 142 | UNTIL | 8E | adc a,(hl) | res 1,(hl) | |
| 143 | ERROR | 8F | adc a,a | res 1,a | |
| 144 | (a) ON | 90 | sub b | res 2,b | outinb |
| 145 | (b) DEFPROC | 91 | sub c | res 2,c | nextreg r,N |
| 146 | (c) ENDPROC | 92 | sub d | res 2,d | nextreg r,a |
| 147 | (d) PROC | 93 | sub e | res 2,e | pixeldn |
| 148 | (e) LOCAL | 94 | sub h | res 2,h | pixelad |
| 149 | (f) DRIVER | 95 | sub l | res 2,l | setae |
| 150 | (g) WHILE | 96 | sub (hl) | res 2,(hl) | |
| 151 | (h) REPEAT | 97 | sub a | res 2,a | |
| 152 | (i) ELSE | 98 | sbc a,b | res 3,b | jp (c) |
| 153 | (j) REMOUNT | 99 | sbc a,c | res 3,c | |
| 154 | (k) BANK | 9A | sbc a,d | res 3,d | |
| 155 | (l) TILE | 9B | sbc a,e | res 3,e | |
| 156 | (m) LAYER | 9C | sbc a,h | res 3,h | |
| 157 | (n) PALETTE | 9D | sbc a,l | res 3,l | |
| 158 | (o) SPRITE | 9E | sbc a,(hl) | res 3,(hl) | |
| 159 | (p) PWD | 9F | sbc a,a | res 3,a | |
| 160 | (q) CD | A0 | and b | res 4,b | ldi |
| 161 | (r) MKDIR | A1 | and c | res 4,c | cpi |
| 162 | (s) RMDIR | A2 | and d | res 4,d | ini |
| 163 | (t) SPECTRUM | A3 | and e | res 4,e | outi |
| 164 | (u) PLAY | A4 | and h | res 4,h | ldix |
| 165 | RND | A5 | and l | res 4,l | ldws |
| 166 | INKEY$ | A6 | and (hl) | res 4,(hl) | |
| 167 | PI | A7 | and a | res 4,a | |
| 168 | FN | A8 | xor b | res 5,b | ldd |
| 169 | POINT | A9 | xor c | res 5,c | cpd |
| 170 | SCREEN$ | AA | xor d | res 5,d | ind |
| 171 | ATTR | AB | xor e | res 5,e | outd |
| 172 | AT | AC | xor h | res 5,h | lddx |
| 173 | TAB | AD | xor l | res 5,l | |
| 174 | VAL$ | AE | xor (hl) | res 5,(hl) | |
| 175 | CODE | AF | xor a | res 5,a | |
| 176 | VAL | B0 | or b | res 6,b | ldir |
| 177 | LEN | B1 | or c | res 6,c | cpir |
| 178 | SIN | B2 | or d | res 6,d | inir |
| 179 | COS | B3 | or e | res 6,e | otir |
| 180 | TAN | B4 | or h | res 6,h | ldirx |
| 181 | ASN | B5 | or l | res 6,l | |
| 182 | ACS | B6 | or (hl) | res 6,(hl) | |
| 183 | ATN | B7 | or a | res 6,a | ldpirx |
| 184 | LN | B8 | cp b | res 7,b | lddr |
| 185 | EXP | B9 | cp c | res 7,c | cpdr |
| 186 | INT | BA | cp d | res 7,d | indr |

| Dec | Character / Control Code / Token | Hex | Z80N Assembler | - after CB | - after ED |
|---|---|---|---|---|---|
| 187 | SQR | BB | cp e | res 7,e | otdr |
| 188 | SGN | BC | cp h | res 7,h | lddrx |
| 189 | ABS | BD | cp l | res 7,l | |
| 190 | PEEK | BE | cp (hl) | res 7,(hl) | |
| 191 | IN | BF | cp a | res 7,a | |
| 192 | USR | C0 | ret nz | set 0,b | |
| 193 | STR$ | C1 | pop bc | set 0,c | |
| 194 | CHR$ | C2 | jp nz,NN | set 0,d | |
| 195 | NOT | C3 | jp NN | set 0,e | |
| 196 | BIN | C4 | call nz,NN | set 0,h | |
| 197 | OR | C5 | push bc | set 0,l | |
| 198 | AND | C6 | add a,N | set 0,(hl) | |
| 199 | <= | C7 | rst 0 | set 0,a | |
| 200 | >= | C8 | ret z | set 1,b | |
| 201 | <> | C9 | ret | set 1,c | |
| 202 | LINE | CA | jp z,NN | set 1,d | |
| 203 | THEN | CB | *modifying prefix* | set 1,e | |
| 204 | TO | CC | call z,NN | set 1,h | |
| 205 | STEP | CD | call NN | set 1,l | |
| 206 | DEF FN | CE | adc a,N | set 1,(hl) | |
| 207 | CAT | CF | rst 8 | set 1,a | |
| 208 | FORMAT | D0 | ret nc | set 2,b | |
| 209 | MOVE | D1 | pop de | set 2,c | |
| 210 | ERASE | D2 | jp nc,NN | set 2,d | |
| 211 | OPEN # | D3 | out (N),a | set 2,e | |
| 212 | CLOSE # | D4 | call nc,NN | set 2,h | |
| 213 | MERGE | D5 | push de | set 2,l | |
| 214 | VERIFY | D6 | sub N | set 2,(hl) | |
| 215 | BEEP | D7 | rst 16 | set 2,a | |
| 216 | CIRCLE | D8 | ret c | set 3,b | |
| 217 | INK | D9 | exx | set 3,c | |
| 218 | PAPER | DA | jp c,NN | set 3,d | |
| 219 | FLASH | DB | in a,(N) | set 3,e | |
| 220 | BRIGHT | DC | call c,NN | set 3,h | |
| 221 | INVERSE | DD | *IX prefix** | set 3,l | |
| 222 | OVER | DE | sbc a,N | set 3,(hl) | |
| 223 | OUT | DF | rst 24 | set 3,a | |
| 224 | LPRINT | E0 | ret po | set 4,b | |
| 225 | LLIST | E1 | pop hl | set 4,c | |
| 226 | STOP | E2 | jp po,NN | set 4,d | |
| 227 | READ | E3 | ex (sp),hl | set 4,e | |
| 228 | DATA | E4 | call po,NN | set 4,h | |
| 229 | RESTORE | E5 | push hl | set 4,l | |
| 230 | NEW | E6 | and N | set 4,(hl) | |
| 231 | BORDER | E7 | rst 32 | set 4,a | |
| 232 | CONTINUE | E8 | ret pe | set 5,b | |
| 233 | DIM | E9 | jp (hl) | set 5,c | |
| 234 | REM / ; | EA | jp pe,NN | set 5,d | |
| 235 | FOR | EB | ex de,hl | set 5,e | |
| 236 | GO TO | EC | call pe,NN | set 5,h | |
| 237 | GO SUB | ED | *modifying prefix* | set 5,l | |
| 238 | INPUT | EE | xor N | set 5,(hl) | |

| Dec | Character / Control Code / Token | Hex | Z80N Assembler | - after CB | - after ED |
|---|---|---|---|---|---|
| 239 | **LOAD** | EF | rst 40 | set 5,a | |
| 240 | **LIST** | F0 | ret p | set 6,b | |
| 241 | **LET** | F1 | pop af | set 6,c | |
| 242 | **PAUSE** | F2 | jp p,NN | set 6,d | |
| 243 | **NEXT** | F3 | di | set 6,e | |
| 244 | **POKE** | F4 | call p,NN | set 6,h | |
| 245 | **PRINT** | F5 | push af | set 6,l | |
| 246 | **PLOT** | F6 | or N | set 6,(hl) | |
| 247 | **RUN** | F7 | rst 48 | set 6,a | |
| 248 | **SAVE** | F8 | ret m | set 7,b | |
| 249 | **RANDOMIZE** | F9 | ld sp,hl | set 7,c | |
| 250 | **IF** | FA | jp m,NN | set 7,d | |
| 251 | **CLS** | FB | ei | set 7,e | |
| 252 | **DRAW** | FC | call m,NN | set 7,h | |
| 253 | **CLEAR** | FD | *IY prefix** | set 7,l | |
| 254 | **RETURN** | FE | cp N | set 7,(hl) | |
| 255 | **COPY** | FF | rst 56 | set 7,a | |

# Appendix B
# Reference

The following sections provide a handy reference of Error Codes and their equivalent Reports, NextBASIC keywords and functions as well as other information discussed so far in a consice form

## Reports and Error Codes

These appear at the bottom of the screen whenever the computer stops executing some function, and explain why it stopped, whether for a natural reason, or because an error occurred.

The report has a brief message explaining what happened and the bank number (not present unless the error occurred in a banked section of program), the line number and statement number within the line where it stopped (A command is shown as line 0. Within a line, statement 1 is at the beginning, statement 2 comes after the first colon or **THEN**, and so on). Some of the codes will have a code number or letter so that you can refer to the tables below. There are two types of error reports: *General* and *Storage System related.*

## General Errors

The behaviour of **CONTINUE** depends very much on the reports. Normally, **CONTINUE** goes to the line and statement specified in the last report, but there are exceptions with reports **0**, **9** and **D**.

Below, there is a table showing all the reports together with the circumstances they can occur.

| Code | Report | Description | Situation |
|---|---|---|---|
| 0 | OK | Successful completion, or jump to a line number bigger than any existing. This report does not change the line and statement jumped to by **CONTINUE**. | Any |
| 1 | NEXT without FOR | The control variable does not exist (it has not been set up by a **FOR** statement), but there is an ordinary variable with the same name. | **NEXT** |
| 2 | Variable not found | For a simple variable, this will happen if the variable is used before it has been assigned to in a **LET**, **READ** or **INPUT** statement or loaded from tape or set up in a **FOR** statement. For a subscripted variable, it will happen if the variable is used before it has been dimensioned in a **DIM** statement or loaded from a storage device. | Any |
| 3 | Subscript wrong | A subscript is beyond the dimension of the array, or there are the wrong number of subscripts. If the subscript is negative or bigger than **65535**, then error **B** will result. | Subscripted variables, substrings |
| 4 | Out of memory | There is not enough room in the computer for what you are trying to do. If the computer really seems to be stuck in this state, you may have to clear out the command line using **DELETE** and then delete a program line or two (with the intention of putting them back afterwards) to give yourself room to manoeuvre with – say – **CLEAR.** | **LET**, **INPUT**, **FOR** , **DIM**, **GO SUB**, **LOAD**, **MERGE**, **BANK**, **PALETTE**, **SPRITE**, **LAYER**, **TILE.** Sometimes during expression evaluation |
| 5 | Out of screen | An **INPUT** statement has tried to generate more than 23 lines in the lower half of the screen. Also occurs with **PRINT AT 22, …, TILE** and **SPRITE.** | **INPUT**, **PRINT AT**, **SPRITE**, **TILE** |
| 6 | Number too big | Calculations have led to a number greater than about $10^{38}$. | Any arithmetic |
| 7 | RETURN without GO SUB | There has been one more **RETURN** than there were **GO SUB**s. | **RETURN** |
| 8 | End of file | | Storage device, etc, operations |
| 9 | STOP statement | After this, **CONTINUE** will not repeat the **STOP**, but carries on with the statement after. | **STOP** |

| Code | Report | Description | Situation |
|---|---|---|---|
| A | Invalid argument | The argument for a function is no good for some reason. | SQR, LN, ASN, ACS, USR (with string argument) |
| B | Integer out of range | When an integer is required, the floating point argument is rounded to the nearest integer. If this is outside a suitable range then error B results. For array access, see also error 3. | RUN, RANDOMIZE, POKE, DIM, GO TO, GO SUB, LIST, LLIST, PAUSE, PLOT, CHR$, PEEK, USR (with numeric argument), PALETTE, BANK, SPRITE, LAYER, TILE, POINT, Array access |
| C | Nonsense in BASIC | The text of the (string) argument does not form a valid expression. | VAL, VAL$ |
| D | BREAK - CONT repeats | BREAK was pressed during some peripheral operation. The behaviour of CONTINUE after this report is normal in that it repeats the statement. Compare with report L. | LOAD, SAVE, VERIFY, MERGE, LPRINT, LLIST, COPY. Also when the computer asks scroll? and you type N, SPACE or STOP[1] |
| E | Out of DATA | You have tried to READ past the end of the DATA list. | READ |
| F | Invalid file name | SAVE with name that is empty or unacceptable (see Chapter 20) | SAVE |
| G | No room for line | There is not enough room left in memory to accommodate the new program line. | Entering a line into the program |
| H | STOP in INPUT | Some INPUT data started with STOP, or – for INPUT LINE – STOP was pressed. Unlike the case with error 9, after error H CONTINUE will behave normally, by repeating the INPUT statement. | INPUT |
| I | FOR without NEXT | There was a FOR loop to be executed no times (e.g. FOR n=1 TO 0) and the corresponding NEXT statement could not be found. | FOR |
| J | Invalid I/O device | | Storage device etc. operations |
| K | Invalid colour | The number specified is not an appropriate value. | INK, PAPER, BORDER, FLASH, BRIGHT, INVERSE, OVER, PALETTE; also after control characters |
| L | BREAK into program | BREAK pressed, this is detected between two statements. The line and statement number in the report refer to the statement before BREAK was pressed, but CONTINUE goes to the statement after (allowing for any jumps to be done), so it does not repeat any statements. | Any |
| M | RAMTOP no good | The number specified for RAMTOP is either too big or too small. | CLEAR, BANK; possibly in RUN |
| N | Statement lost | Jump to a statement that no longer exists. | RETURN, NEXT, CONTINUE |
| O | Invalid stream | | Storage device, etc, operations |
| P | FN without DEF | An attempt was made to call a function with FN that has not been defined with a matching DEF FN statement. | FN |
| Q | Parameter error | Wrong number of arguments, or one of them is the wrong type (string instead of number or vice versa). | FN |
| R | Tape loading error | A file on tape was found but for some reason could not be read in, or would not verify. | VERIFY, LOAD or MERGE |
| d | Too many parentheses | Too many parentheses around a repeated phrase in one of the arguments. | PLAY |
| i | Invalid device | The storage device specified does not exist | |
| k | Invalid note | PLAY came across a note or command it didn't recognise, or a command which was in lower case. | PLAY |
| l | Too big | A parameter for a command is an order of magnitude too big. | PLAY |

1 *STOP cannot normally be entered in NextBASIC as a token; this is retained for compatibility and does work when you switch to 48K mode*

| Code | Report | Description | Situation |
|---|---|---|---|
| m | Note out of range | A series of sharps or flats has taken a note beyond the range of the sound chip. | PLAY |
| n | Out of range | A parameter for a command is too big or too small. If the error is very large, error L results | PLAY |
| o | Too many tied notes | An attempt was made to tie too many notes together | PLAY |
| | Invalid mode | The mode specified does not exist | LAYER |
| | Direct command error | An attempt was made to execute a command within a program that's meant to be executed directly from the command line or to RUN a procedure definition (DEFPROC) | DEFPROC, ERASE, LINE, LINE MERGE, BANK LINE MERGE |
| | Loop error | Occurs in REPEAT...REPEAT UNTIL loops where a matching REPEAT UNTIL or REPEAT cannot be found. | REPEAT...REPEAT UNTIL, WHILE |
| | No DEFPROC | A PROC was found without a matching DEFPROC...ENDPROC block | PROC |
| | No ENDIF | An ELSEIF was found without a matching ENDIF | IF...ELSEIF...ENDIF |
| | No label | A referenced label, does not exist | |

## Storage Device Related Errors

The following are reports generated by *NextZXOS* for storage device errors. Those marked in the left-hand column with RIC may be followed by the options Retry, Ignore or Cancel?

Some reports may occur with the code(s) shown or without them.

| Code | Report | Description |
|---|---|---|
| e | Already exists | The destination filename or directory already exists. Also occurs when attempting to map a drive letter that is already mapped to another device. |
| | Bad file number | An attempt was made to operate on a file which has not been opened. It is unlikely that this error will ever be seen. |
| f | Bad filename | The filename used does not conform to the filename requirements for the filesystem. |
| | Bad parameters | One of the values provided is out of range. |
| | Code length error | Trying to load a CODE file from the storage device that is longer than the value given on the LOAD command. |
| | Dest can't be wild | Trying to give a wildcard file specification for the destination file in a COPY command when the source also contains wildcard characters. In this case, the destination can only be a drive letter. |
| | Dest must be path | The source filename in a COPY command contains wildcard characters, but the destination is only a single file name. In this case, the destination can only be a path. |
| | Dir full | Unable to add further entries to the directory, or unable to remove a directory because it contains files or subdirectories. |
| RIC | Disk changed | The disk in the drive has been changed without properly REMOUNTing. |
| RIC | Disk error | An error has occurred accessing a storage device. If this error persists it may indicate that the device is faulty. |
| | Disk full | Saving or copying files to a storage device has used up the free space. The CAT command can be used to check that there is sufficient free space before attempting such an operation. This may leave a partly-written file if there was only space for some of it. This part should be erased, as any attempt to use it will fail. |
| | Dot command error | The error that was trapped by ON ERROR was generated by a dot command. This is seen only when ERROR is used to cause the last trapped error. |
| | End of file | An attempt has been made to read a byte past the end-of-file position. |
| g,h | File not found | The filename specifies a file that does not exist. |
| | Fragmented – use .DEFRAG | The file is split into parts across the disk. Defragment it using the .DEFRAG dot command. |
| | In use | An attempt has been made to unmap or re-map a drive that has files open on it, or to access a file that is already open for another purpose. |
| | Invalid attribute | The attribute character following + or - in a MOVE command is not P, S or A (or there is more than one character after the +/-). |
| | Invalid device | The physical device specified does not exist. |
| | Invalid drive | A drive letter that does not exist has been specified. |
| | Invalid partition | The partition specified does not exist, or is the wrong type. |
| | Invalid path | The path specified does not exist |
| | No rename between drives | An attempt has been made to use the MOVE command specifying source and destination filenames that are on different drives. |

| | | |
|---|---|---|
| | No swap partition | An application attempted to access a swap partition, but couldn't find one. Create a new swap partition with **.MKSWAP** and try again. |
| | Not bootable | An attempt has been made to boot a disk image without a boot sector or boot program. |
| | Not implemented | An attempt was made to access a facility which isn't available. |
| RIC | Not ready | The storage device was not ready. This usually happens because it has been removed. |
| | Out of handles | There aren't enough handles left to perform the current operation. Unmap a drive and try again. |
| | Partition open | The partition you are trying to delete or map is already mapped to a drive. |
| RIC | Read only | An attempt has been made to write to a file or storage device which is read-only or has been write-protected. |
| RIC | Seek fail | The device is unable to locate the sector that has been requested. If this error persists it may indicate that the device or disk image is faulty. |
| | Too big | An attempt has been made to write a file that is too large for the filesystem (greater than 8MB for +3DOS filesystems, 2GB on FAT16 or 4GB on FAT32). |
| RIC | Unsuitable media | The device or disk image is formatted in a way that cannot be handled. |
| b | Wrong file type | Trying to **LOAD** a file of the wrong type (eg trying to load a **CODE** file as a *NextBASIC* program). |

## NextBASIC Keywords and Functions

The following is a list of all *NextBASIC* keywords in alphabetical order with a short description regarding their function.

| Keyword | Meaning |
|---|---|
| **BANK 1346 FORMAT** | Reserve banks 1,3,4,6 for use by the RAMdisk again. |
| **BANK 1346 USR** | Allow banks 1,3,4,6 to be used by the BANK command. |
| **BANK** m **COPY TO** n | Copy the contents of bank m to bank n |
| **BANK** m **DPOKE** o, list... | Double POKE a sequence of comma-separated values starting at offset o in bank m. |
| **BANK** m **ERASE** [o, l,] [v] | Fill bank m's optional l bytes (all if not specified) at optional offset o (0 if not specified) with value (zero is used if value not specified). |
| **BANK** m **CLEAR** | Marks bank m as free for use by other parts of the system. |
| **BANK** m **COPY** o, l **TO** n,o2 | Copy l bytes starting at offset o in bank m to offset o2 in bank n. |
| **BANK** m **GOSUB** n | GOSUB line n in bank m. To GOSUB the main program from a banked section, use m=255. See also RETURN and GOSUB. |
| **BANK** m **GOTO** n | GOTO line n in bank m. To GOTO the main program from a banked section, use m=255. |
| **BANK** m **LAYER** o\|x,y,w,h **TO** [rop] x,y,w,h\|o | Copies data to \| from the screen (in the current mode) from \| to offset in bank m. [rop] is an optional symbol modifier which affects how the data is copied. |
| **BANK** m **LINE** x,y | Copies lines x to y inclusive from the main program to bank m. |
| **BANK** m **LIST** [n\|**PROC** name()] | List lines (optionally from line n or procedure named name) in bank m. |
| **BANK** m **MERGE** | Copy all lines back from bank m into the main program. |
| **BANK** m **POKE** o, list... | POKE a sequence of comma-separated values starting at offset o in bank m. |
| **BANK** m **PROC** name ([expressionlist]) [**TO** paramlist] | Call a procedure in bank m. To call a procedure in the main program from a banked section, use n=255. See also DEFPROC. |
| **BANK** m **RESTORE** n | Set the DATA pointer to line n in bank m |
| **BANK NEW** var | Reserves the next available free bank number and assigns it to the numeric variable var |
| **BEEP** x, y | Sounds a note through the loudspeaker for x seconds at a pitch y semitones above middle C (or below if y is negative). |
| **BORDER** m | Sets the colour of the border of the screen. |
| **BRIGHT** n | Sets brightness of characters subsequently printed. n=0 for normal, 1 for bright. 8 for transparent.Error K if n not 0, 1 or 8 |
| **CAT** [#n,] [[filespec [**EXP**]]\|**TAB**\|**ASN**] | Produces an alphanumerically sorted catalog of files on screen or to an optional stream n from the default drive or according to the optional filespec in standard or EXPanded form. With the optional TAB and ASN modifiers produces information regarding partitions and drive letter assignments. |
| **CD** filespec | Change the current drive and/or directory to the one specified in filespec. |
| **CIRCLE** x, y, z | Draws an arc of a circle, centre (x,y), radius z |
| **CLEAR** [n] | Deletes all variables, freeing the space they occupied. Does RESTORE and CLS, resets the PLOT position to the bottom left-hand corner and clears the NextBASIC Return stack. Optional address n attempts to change the RAMTOP to that address |
| **CLOSE** #n | Marks stream n as being unattached to any channel. |
| **CLS** | (Clear Screen). Clears the display of the current layer |

| Keyword | Meaning |
|---|---|
| CONTINUE | Continues the program, starting where it left off last time it stopped with report other than 0. |
| COPY | Sends (dumps) a copy of the screen display to a ZX Printer or compatible. |
| COPY u TO SCREEN$ | Displays the contents of a file defined by filespec u on the screen. Control characters (tabs, line feeds, etc.) are replaced by spaces. |
| COPY u1 TO u2 | Copies file(s) defined by filespec u1 to the destination defined by filespec u2 |
| DATA list ... | Part of the DATA list. Must be in a program, otherwise has no effect. |
| DEF FN ? (?1,..., ?k)=e | User-defined function definition; must be in a program. Each of ? and ?1 to ?k is either a single letter or a single letter followed by $ for string argument or result.Takes the form DEF FN a()=e if no arguments. |
| DEFPROC name ([paramlist]) | Defines a procedure, where name follows the same naming rules as standard numeric variables. paramlist is an optional list of up to 8 variable names (simple strings, numeric variables or integer variables, but not arrays of any type). See ENDPROC. |
| DIM #n,var | Returns the extent (or size) of stream n and stores it in variable var. |
| DIM ?( n1 , . . . ,nk ) | Deletes any array or string with the name ?, and sets up an array of characters or numbers with k dimensions n1 ,...,nk. Initialises all the values to . This can be considered as an array of strings of fixed length nk , with k-1 dimensions n1,...,nk-1 . An array is undefined until it is dimensioned in a DIM statement. |
| DRAW x,y [,z] | Draws a line from the current plot position moving x horizontally and y vertically relative to it while turning through an optional angle z |
| DRIVER drid,callid[,n1[,n2]] [TO var1[,var2[,var3]]] | Call function callid in driver drid, where n1 and n2 are optional values to pass to the driver, and var1, var2 and var3 are optional variables to receive results back from the driver. |
| ELSE | See IF ... THEN ... ELSE |
| ENDPROC [= expressionlist] | Ends execution of a procedure defined with DEFPROC and returns up to 8 local values via the optional expressionlist to the calling PROC command. |
| ERASE [m,n] | Erases the entire NextBASIC program and leaves variables intact. If specified with the optional m and n parameters, erases all program lines between m and n inclusive. |
| ERASE filespec | ERASES all files specified by filespec. Cannot erase entire drives |
| ERROR [TO e[,l[,s[,b]]]] | Regenerate the last error that was trapped by an ON ERROR command and store it in optional variables e, l, s, b (for error code, line, statement number and bank) |
| FLASH n | Defines whether characters will be flashing or steady. |
| FOR ?=x TO y [STEP z] | Deletes any simple variable ? and sets up a control variable with value x, limit y, optional step z (or 1 if STEP is not defined), and looping address referring to the statement after the FOR statement. See NEXT. |
| GO TO n | Jumps to line n (or, if there is none, the first line after that). See also BANK...GO TO. |
| GO TO #n, m | Sets the current position of stream n to m. |
| GOSUB n | Pushes the line number of the GOSUB statement onto a stack; then as GO TO n. See also RETURN and BANK...GOSUB. |
| IF x THEN y [: ELSE z] | If x is true (non-zero) then statement list y is executed, otherwise optional statement list z is executed. ELSE must be on the same line as IF. |
| INK n | Sets the ink (foreground) colour of characters subsequently printed. |
| INPUT [#n] [LINE] inputitems | INPUTs inputitems from the keyboard or optional stream n. Optional LINE modifier strips the quotes from the input items |
| INVERSE n | Inverts the next printed character(s) from INK to PAPER |
| LAYER AT x,y | Sets the display offset for the top-left of the screen for the current layer to x,y. |
| LAYER BANK n,m | (Layer 2 only). Set current banks n...n+2 as frontbuffer (to be displayed) and banks m...m+2 as backbuffer (for rendering). |
| LAYER CLEAR | Resets all layer information to the default values. Resets memory banks, mode, layer 2 enable, layer offsets and layer ordering. Also done by NEW |
| LAYER DIM x1,y1,x2,y2 | Sets the clip window for the current layer from (x1,y1) to (x2,y2). Areas of the layer outside this window are not visible. |
| LAYER ERASE x,y,w,h[,f] | Fill region width w pixels, height h pixels, top-left corner x,y with optional value f. If f is not specifed, 0 is used. |
| LAYER m[,n] | Selects the screen layer m and optional mode m. |
| LAYER OVER n | Sets sprite/layer SLU ordering |
| LAYER PALETTE n [BANK m,o ]\| n,i,v | Switch to using palette n (0 or 1) for the current layer and optionally sets palette from bank m, offset o -or- defines index I for palette n as 9-bit colour v |
| LET [%]v = [%]e | Assigns the value of [optionally integer] expression e to the [optionally integer] variable v. LET cannot be omitted. |
| LINE start, step\|m,n TO mm,nn | Either renumbers an entire NextBASIC program starting with line start with an increment of step -or- a section of the NextBASIC program, beginning with line m and ending with line n, with the new starting line number mm and incrementing by nn. |
| LINE MERGE first,last | Merges lines from first to last into a single line (separated by colons). Can only be used as a direct command, not within a program. |
| LIST [[#n],] [m\|PROC name] | Lists the current program to the screen or optional stream number starting with optional line m -or- PROC name. See also BANK...LIST and LLIST |
| LLIST [m] | Like LIST but using the printer |

| Keyword | Meaning |
|---|---|
| **LOAD** filespec [**BANK** m[,o[,n]]\|**CODE** m[,n]\|**DATA** arrayspec\|**LAYER**\|**SCREEN** $] | If filespec is a drivespec: Makes the named drive the current default input device for all subsequent disk operations (COPY, ERASE, MOVE etc.). If the drive letter specified is 'T:', then all subsequent LOADs will default to tape else loads a NextBASIC program into memory. With optional modifier BANK it loads the file as binary data into bank m at optional offset o and optional length n. Optional modifer CODE does the same at address m an optional length n. Optional modifier DATA loads stored data into the array specified by arrayspec. Optional modifier LAYER attemps to load a screen into the current layer while SCREEN$ does the same for Layer 0 screens. See also SAVE, MERGE, VERIFY. If a drive letter is not specified in the filespec, the default drive will be used. |
| **LOCAL** variablelist | Defines a local variable inside a procedure defined with DEFPROC or a subroutine called with GOSUB. One local command accepts up to 256 variable names, and multiple LOCAL commands may be used. |
| **LPRINT** | Like PRINT, but using the printer. |
| **MERGE** filespec | Like LOAD filespec but does not delete old program lines and variables except to make way for new ones with the same line number or name. If a drive letter is not specified, the default drive will be used. |
| **MKDIR** filespec | Create a new directory/folder specified by filespec on the current storage device. If filespec includes a drivespec then that drive will be used |
| **MOVE** filespec1, filespec2 | Renames and/or moves a file defined in filespec1 to filespec2 within the same drive. |
| **MOVE** filespec **TO** attribute | Sets or resets attributes for the file(s) defined by filespec |
| **NEW** | Starts the NextBASIC system afresh, deleting any program and variables, and using the memory up to and including the byte whose address is in the system variable RAMTOP. The system variables UDG, P RAMT, RASP and PIP are preserved. Returns control to the Startup menu, but does not erase files held on drive M: (the RAMdisk). |
| **NEXT** ? | Finds the control variable ?, adds its step to its value and jumps to the looping statement or exits if the limit has been reached. See also FOR. |
| **NEXT** #n,v | Gets the next character of input from stream n and stores it in the variable v. |
| **ON ERROR** [statementlist] | Turns off error trapping or if used with the optional statement list, the statementlist will execute where an error report would normally appear. |
| **OPEN** #n,channelspec | Allows stream number to be attached to the channel identified by channelspec. |
| **OUT** m,n | Outputs byte n at I/O port address m. |
| **OVER** n | Controls overprinting for characters subsequently printed. |
| **PALETTE CLEAR** | Resets all palettes and related settings to defaults. This is also done by NEW. |
| **PALETTE DIM** n | Sets palette type as 8 or 9 bit. |
| **PALETTE FORMAT** n | Enables the EnhancedULA extended palette with n INKs (1,3,7,15,31,63,127 or 255) or disables it (0) |
| **PALETTE OVER** n | Sets the global transparency colour to n (default value is 227). |
| **PAPER** n | Like INK, but controlling the paper (background) colour. |
| **PAUSE** n | Stops computing and displays the display file for n frames. |
| **PLAY** f1[,f2,...f9] | Interpret up to nine command strings and play them simultaneously. |
| **PLOT** x,y | Draws a pixel in the current INK colour (subject to OVER and INVERSE) at the x,y coordinate of the current layer. |
| **POINT** x,y **TO** var | Checks the pixel on the current layer at (x,y) and stores the value in variable var. |
| **POKE** a,valuelist | POKEs the list of values in valuelist to memory map address a. Se also BANK POKE. |
| **DPOKE** addr,valuelist... | Double POKEs the list of values in valuelist to memory map address a. Se also BANK DPOKE. |
| **PRINT** [#n,] [**AT** x,y;] items | Output items to the display or optionally to stream n. Optional AT modifier positions the output at x,y |
| **PRINT POINT** x,y | Set the print position to pixel coordinates x,y. |
| **PROC** name (expressionlist) [**TO** paramlist] | Call procedure defined with DEFPROC. The number of expressions and each of their types must match those defined in the DEFPROC, otherwise a Q Parameter Error report will be generated. TO paramlist will copy return values declared by ENDPROC to up to 8 variables. |
| **PWD** [#n] | Prints the current working directory to the screen, or the specified stream number. |
| **RANDOMIZE** [n] | Sets the system variable (called SEED) used to generate the next value of RND. If optional n =0 or blank SEED is given the value of another system variable (called FRAMES). |
| **READ** v1, v2 ,... vk | Assigns to the variables using successive expressions in the DATA list. |
| **REG** n,v | Sets Next Register n with value v. |
| **REM** ...<br>; ... | Remark. No effect. ' . . . ' can be any sequence of characters except ENTER. |
| **REMOUNT** | Reinitialises the filing system, following a change of SD card. |
| **REPEAT**<br>statementlist<br>[**WHILE** y statementlist2]<br>**REPEAT UNTIL** x | Statement or statements in statementlist and statement list2 are repeated until x is true. The loop is terminated skipping statementlist2 if y evaluates to false |
| **RESTORE** [n] | Restores the DATA pointer to the first DATA statement in line optional line n or to the first DATA statement. |

| Keyword | Meaning |
|---|---|
| **RETURN** | Takes a reference to a statement off the NextBASIC Return stack, and jumps to the line after it. See also GOSUB and BANK GOSUB. |
| **RETURN** #n,var | Takes the current position of stream n and stores it in variable var. |
| **RMDIR** filespec | Removes an already empty folder as specified by filespec. |
| **RUN** [n] | CLEAR, and then GO TO optional line n or to the first line of the program |
| **RUN AT** speed | Changes the speed of the ZX Spectrum Next. |
| **SAVE** filespec [**LINE** n\|**BANK** m[,o[,n]]\|**CODE** m[,n]\|**DATA** arrayspec\|**LAYER**\|**SCREEN** $] | If filespec is a drivespec: Makes the named drive the current default input device for all subsequent disk operations (COPY, ERASE, MOVE etc.). If the drive letter specified is 'T:', then all subsequent SAVEs will default to tape else saves a NextBASIC program into memory with optional modifier LINE n that instructs subsequent LOAD operations to start executing the program from line n. With optional modifier BANK it SAVES the file as binary data from bank m at optional offset o and optional length n. Optional modifer CODE does the same at address m an optional length n. Optional modifier DATA saves the array specified by arrayspec. Optional modifier LAYER saves the current layer's display while SCREEN$ does the same for Layer 0 screens. See also LOAD, MERGE, VERIFY. If a drive letter is not specified in the filespec, the default drive will be used. |
| **SPECTRUM** [filespec\|**ATTR** n\|**BRIGHT** n\|**CHR$** n\|**FLASH** n\|**INK** n\|**PAPER** n\|**SCREEN$** n,t] | Sets the 128K ROM into Spectrum 48K compatibility mode. Optional filespec defining a 48K/128K/ZX80 and ZX81 snapshot loads and executes it. Optional ATTR modifier, sets the colour scheme of NextBASIC Editor. Optional BRIGHT modifier, sets the BRIGHT bit of the colour scheme of NextBASIC Editor. Optional CHR$ modifier, changes the mode to 32/64/85 columns. Optional FLASH modifier, sets the flash bit of the colour scheme of NextBASIC Editor. Optional INK modifier sets the ink colour of the NextBASIC Editor while the PAPER modifiers sets the paper colour of the NextBASIC Editor. The SCREEN$ modifier adjusts the screensaver. |
| **SPRITE BANK** b [,o,p,n] | Defines all 64 sprite patterns using the 16K of data (256 bytes per sprite) in bank b or with optional values o,p,n defines n sprite patterns starting with pattern p located at offset n. |
| **SPRITE BORDER** n | Enable (n=1) or disable (n=0) sprites over the border |
| **SPRITE CLEAR** | Resets the sprite attributes and global settings to defaults. This is also done by NEW. |
| **SPRITE DIM** x1,y1,x2,y2 | Sets the clip window for sprites from (x1,y1) to (x2,y2). |
| **SPRITE PALETTE** n [**BANK** m,o ]\| n,i,v | Switch to using palette n (0 or 1) for the Sprite System and optionally sets palette from bank m, offset o -or- defines index I for palette n as 9-bit colour v |
| **SPRITE PRINT** n | Enable (n=1) or disable (n=0) sprites. |
| **SPRITE** s,x,y,i,f | Set sprite s to image i, position (x,y) with flags f. |
| **STOP** | Stops the program with report 9. See also CONTINUTE |
| **TILE** w,h \|**AT** x,y [**TO** x2,y2] | Draws a section of the screen from a tilemap. Optional AT specifies tile offset x,y in the tilemap and optional TO specifies ending tile offset x2,y2 |
| **TILE BANK** n | Define bank n as containing the tiles (up to 4 banks n..n+3 if 16x16 tiles). |
| **TILE DIM** n,offset,w,tilesize | Define bank n as containing the tilemap, starting at offset offset in the bank. The tilemap is width w (1-2048) and uses 8x8 (tilesize=8) or 16x16 (tilesize=16) tiles. |
| **VERIFY** filespec | Like LOAD (from tape), but the tape information is not loaded into RAM – instead, it is just compared against what is already in RAM.If the filespec is a drive letter, then sets the default drive. Only applicable to tape |

The following is a list of all *NextBASIC* functions in alphabetical order with a short description regarding their purpose:

| Function | Meaning |
|---|---|
| **ABS** x | Absolute Value of x |
| **ACS** x | Arccosine of x in radians |
| **ASN** x | Arcsine of x in radians |
| **ATN** x | Arctangent of x in radians |
| **ATTR** (x,y) | A number whose binary form codes the attributes of line x, column y on the display |
| **CHR$** n | The character whose code is n, rounded to the nearest integer. |
| **CODE** f | The code of the first character in string f (or 0 if f is the empty string). |
| **COS** x | Cosine (x in radians). |
| [**BANK** n] **DPEEK** a | Reads a double-byte (16 bit word) from memory address a or bank n offset a. |
| **EXP** x | Returns the natural exponential function of e to the power x. |
| **FN** a() | FN followed by a letter calls up a user-defined function (see keyword DEF FN). |
| **IN** n | The result of inputting at processor level from port n |
| **INKEY$** | Reads the keyboard. |
| **INT** x | Returns the Integer part of floating point expression x (Always rounds down) |
| **INT** { x } | Returns an unsigned 16-bit integer expression, from any floating point expression x |
| **LEN** string | Returns the length of string |
| **LN** x | Natural logarithm (to base e). |

| Function | Meaning |
|---|---|
| [BANK n] PEEK o | Returns the byte at address o or if used with the optional BANK, the byte at offset o of bank n |
| [BANK n] PEEK$ (o,len\|t) | Reads memory region of length len stored in the addresses beginning with o and stores it in a string –or– Reads the string terminated with a user specified terminator t beginning with address o. With the optional BANK reads offset o of bank n. |
| PI | Returns and approximation of π (3.14159265...) |
| POINT (x,y) | Retruns 1 if the pixel at (x,y) is ink colour. 0 if it is paper colour. |
| REG n | Reads state of Next Register n |
| RND [n\|i] | Returns the next pseudorandom number n in the range from 0 to 1 –or– the next pseudorandom integer number in the range of 0 to i-1 |
| SCREEN$ (x, y) | Returns the character that appears, either normally or inverted, on the display at line x, column y. |
| SGN x | Signum; the sign (-1 for negative, 0 for zero or +1 for positive) of x. |
| SGN {i} | Returns a signed 16-bit integer from integer expression i |
| SIN x | Returns the sine of x in radians. |
| SQR x | Returns the square root of x. |
| STR$ x | Returns the string of characters that would be displayed if x were printed. |
| TAN x | Returns the tangent of x in radians. |
| [BANK n] USR o | Calls the machine code subroutine whose starting address is o. With optional BANK does the same for offset o in bank n. On return, the result is the contents of the bc register pair. |
| USR l | The address of the bit pattern for the user-defined graphic corresponding to character l. |
| VAL f | Evaluates string f (without its bounding quotes) as a numerical expression. |
| VAL$ f | Evaluates string f (without its bounding quotes) as a string expression. |

## The Decimal System

Most European languages count using a more or less regular pattern of tens – in English, for example, although it starts off a bit erratically, it soon settles down into regular groups:

twenty, twenty one, twenty two, . . . twenty nine
thirty, thirty one, thirty two, . . . thirty nine
forty, forty one, forty two, . . . forty nine

This follows from using Arabic numerals, which have ten symbols **0 – 9**, in a placeholder system where the position of each digit is multiplied by a power of ten. The reason for using ten as the basis of numbers is that we happen to have ten fingers.

## The Binary System

Instead of using the *decimal* system, with *ten* as its *base*, computers use a system called *binary*, based on two values **0** and **1**. Like humans have ten fingers, computer circuits have two states; low-voltage or off (**0**) and high-voltage (**1**). The two binary digits are called *bits*, and a bit is either **0** or **1**. Computers therefore write **10** to represent **2**, **100** to represent **4**, **1000** to represent **8**, and so on for the powers of **2**.

It is customary to "pad out" binary numbers with leading zeroes so that they always contain at least four bits, called a *nibble* – for example, **0000**, **0001**, **0010**, **0011** (representing **0** to **3** *decimal*). The reason for doing this is that it makes it easy to represent long binary numbers more compactly using hexadecimal as we will see further below.

Throughout this manual we've written binary numbers either with the suffix of a lower case **b** or with the prefixes of @ and **BIN** as supported by the *NextBASIC* Integer expression evaluator.

Regardless of how useful it is to write numbers in the way computers understand them, we have the obvious problem of representing them on paper; it's much easier for us to write and understand

**65535** + **65534** than **1111111111111111b** + **1111111111111110b**.

## The Hexadecimal System

Binary numbers quickly become unwieldly because even modest quantities require long strings of **0**s and **1**s to represent them. This is a natural result of only using two symbols to

represent each digit. *Hexadecimal* (or hex for short) was adopted to easily and compactly represent binary numbers. Hexadecimal is a *base 16* numbering system with 16 symbols. 0 through 9 are used for the first ten symbols, representing decimal values 0 – 9, and the last six symbols are A, B, C, D, E, F representing decimal values 10 – 15. What comes after F? Just as in decimal we write 10 for ten, in hexadecimal we write 10 for sixteen since each position is associated with a power of 16.

The reason why hexadecimal is so well suited to representing binary numbers is that sixteen is a *power of 2*. This means binary digits can be grouped together and directly converted to a hexadecimal digit. Since sixteen is the *fourth power of 2*, four binary digits – *a nibble* – can be represented by a single hexadecimal digit. Conversion between binary and hexadecimal can then be done by sight and hexadecimal becomes a quick way to represent large binary quantities as well as an easy way to visualize bit patterns.

The table below shows the correspondence between binary, hexadecimal and decimal values:

| Binary | 0000 | 0001 | 0010 | 0011 | 0100 | 0101 | 0110 | 0111 | 1000 | 1001 | 1010 | 1011 | 1100 | 1101 | 1110 | 1111 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Hexadecimal | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
| Decimal | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |

To convert hex to binary, change each hex digit into a nibble (four bits), using the table above. Conversely, to convert binary to hex, divide the binary number into nibbles, starting on the right, and then change each group into the corresponding hex digit.

Throughout this manual, we've written hexadecimal numbers suffixed by a lower case letter h or prefixed by $ as the latter notation is the one supported by the *NextBASIC Integer Expression* evaluator.

## Bits, Bytes and Words

The bits inside the computer are mostly grouped into sets of eight – these are called *bytes*. A single byte can represent any number from 0 to 255 decimal (11111111b or FFh). A single byte can also represent any character in the ZX Spectrum Next character set. Its value can be written with two hex digits.

Two bytes can be grouped together to make what is called a *word*. A word can be written using sixteen bits or four hex digits, and represents a number from 0 to 65535 decimal.

A byte is always eight bits, but words vary in length from computer to computer. In Sinclair computer tradition, 16-bit numbers are called words while 32-bit numbers are called long words.

*Setting a bit* means making a specific bit 1. *Resetting a bit* means making a specific bit 0. In digital logic, there is also a concept of “active low” and “active high”. This means a signal becomes active when it is 0 or 1 respectively. The Z80n has an MREQ (or /MREQ) signal, for example. This is an “active low” signal; to distinguish them from “active high” signals, we usually write active low signals with a bar over their names (Or prefix them with a forward slash /). This means the Z80n indicates a memory cycle by making MREQ 0.

## Using Binary and Hex in NextBASIC

Our first introduction to binary and hex was in *Chapter 7* which introduced Integer Expressions. *Chapter 14* introduced the use of the BIN keyword. *Chapter 16* showed us how useful binary was in defining colours with the PALETTE keyword while *Chapters 23* and *24* with the introduction of binary bitmasks for the REG and OUT keywords and the memory address space showed the usefulness of hexadecimal.

In reality many keyword parameters are binary; As an example ATTR and RUN AT's decimal parameters are really decimal “translations” of the bits that are being set inside the computer's memory or the Next Registers that these keywords control.

# Appendix C

## Machine Personalities

### Overview

Your ZX Spectrum Next computer is unique in the fact that unlike other computers that emulate older machines using software, it changes its actual hardware to reflect the hardware of an older ZX Spectrum model. This fact, is what allows it to achieve almost 100% compatibility with older models whereas even a ZX Spectrum 128K for example couldn't run a lot of software originally made for the 48K.

The technology that makes all this possible is contained within a very large reconfigurable logic device called a *Field Programmable Gate Array* (FPGA).

For all purposes, once your ZX Spectrum Next goes into an older ZX Spectrum model personality, it's almost identical to that model internally. Moreover, since FPGAs can be made into almost any conceivable kind of digital circuit using a *Hardware Description Language* (HDL), your ZX Spectrum Next can become other machines using different CPU models; this is what we call *multicore* capability.

Before we examine what machine personalities are available on the ZX Spectrum Next, it is a good idea to start with learning about how to update the machine's core(s), firmware and system software.

### The Cores and their update procedures

The ZX Spectrum Next is primarily a ZX Spectrum computer; its main core will always be one of a ZX Spectrum compatible machine (albeit with many extra features) however since it is also a multicore machine, it has two separate (but very similar) procedures to update its main core and a third one for additional cores[1].

Let us first clarify a few things about what a core is and what it isn't. A core is a bitstream written in an HDL, "compiled" for the specific models of Xilinx™ FPGA the ZX Spectrum Next Issues 2 and 4 use and stored onboard a serial flash rom IC on the ZX Spectrum Next board. It contains all the logic that allows the FPGA device to reconfigure itself into the individual components that make up a ZX Spectrum Next. Every time you turn on your ZX Spectrum Next the core gets tranferred to the FPGA almost instantaneously. It doesn't get etched permanently inside the FPGA; instead the FPGA is empty every time the computer gets powered up.

We may be talking about one ZX Spectrum Next core but in reality due to the two production mainboards in existence (each with a different model of a Xilinx-brand FPGA), there are four; there are two *Anti Brick* cores (AB) and two regular cores you get with every new System/Next™ update. The AB cores serve two purposes: The main is to perform the inital machine startup and the secondary is to protect you from a botched attempt to flash a new core into the system's flash rom (hence the name Anti Brick[2]).

Every time the computer starts it transfers the AB core onto the FPGA, then the AB core loads the Firmware file from the root folder of the System/Next™ distribution and that in turn loads the regular core into the FPGA, then loads the appropriate configuration and finally starts the machine.

The AB core does not get updated; only the regular core is. The AB core needs a special procedure which is done at the factory to get updated so it won't be covered here.

There are two methods of updating the regular core; the first one is the normal one, and the one you should use while the second one is reserved only if told so by the release

1 *SpecNext Ltd does not offer additional cores at the time of writing; 3rd party cores are the responsibility of their respective authors*

2 *Bricking is a term used for a failed update in digital electronics that leaves a device unusable; in other words unmovable as a "brick".*

notes of a System/Next™ distribution or because your update somehow failed (for example lost power while updating).

The regular core (for both Issue 2 and Issue 4 mainboards) is contained within a file named TBBLUE.TBU. In order to update the flash rom you need to place it on the root folder of your SD Card together with the file containing the firmware: TBBLUE.FW. Both of these files need to be present for a successful update. Regular operations, however , require only the TBBLUE.FW file to be present at all times in the root of your System/Next™ distribution. Regardless of the update method you need to have them both so make a note for that.

Just placing a TBBLUE.TBU file on the root of the card won't update the core; there are additional steps you need to take. Let's examine the two update options below.

## Regular Core update

The regular core update method is quite easy. After you've made sure you have the TBBLUE.FW and TBBLUE.TBU on the root folder of your card, press and hold U on your keyboard and while doing that, *long*[3] press the RESET button. Do not release the U key until you see the following screen (Note that if you have a KS1 ZX Spectrum Next, an N-Go computer or a KS1 dev board your Board Id will say ZX Next Issue 2):




*Fig. 50 – Core update screen*

Release U and then press Y. The updater will first calculate the checksum of the core bistream; once it finds everything is OK, it will start upgrading; first erasing the Flash ROM and then, once done successfully, writing the core bistream from TBBLUE.TBU in its place. Once the procedure has finished, you will receive an: Updated! Turn the power off and on. message. Remove the power and if using an HDMI display, the display cable as well. Wait a few moments and then reconnect everything. The machine should restart with the new core.

## AB Core update

If the process failed somehow; or if you're so instructed by the accompanying notes of your System/Next™ distribution, you can do an *AB core update* to remedy the situation. This is a bit more complicated and it's made so as to avoid entering this mode by mistake.

To enter AB core update; you will need to power off your machine, then press and hold the NMI and Drive buttons together (on the side of the computer) and while doing that reat-

3 More than 1 second

tach the power cable. Wait a few moments then release both keys. You should see the following screen:




*Fig. 51 – AB Core update screen*

If the display is blank, press **F3** on the keyboard. Note that due to AB core using the **NMI** and **Drive** buttons you cannot press **F3** using the **NMI** + **3** shortcut so you must have a PS/2 keyboard for that.

The display could be blank because the AB core works at 60 Hz in VGA mode only so if your display cannot "lock" onto that mode and you have no PS/2 keyboard to attach, you will need to do a so-called "blind update". You can still press **y** and more than likely the update will finish however if you have no display, the preferred method of performing said update is by pressing the **NMI** button once which in AB core update is a shortcut for **y** while the **Drive** button is a shortcut for **n**. If you do perform a "blind update" you should allow the machine adequate time to finish.

**Please note that on an Issue 4, the average AB update time is 15 minutes from the time you press y so allow about 20 minutes before turning the power off.**

## Multicore (Extra Cores) update

The Extra Cores update deals with the optional third party cores the ZX Spectrum Next accepts. The process is similar with two exceptions. You will need a file called **CORExxx.BIT** where **xxx** is a number from **001** to **031** instead of the **TBBLUE.TBU** placed in the root folder of your **System/Next™** distribution and you enter it by pressing and holding **C** instead of **U** while in *NextZXOS*. Every other step is exactly the same. Your 3rd party core will come with instructions on what to do and how to start the core. Generally speaking, files specific to that core go under the **c:/machines/** folder, into one subfolder specific to that core. So if, for example, a QL core was released, you would find all pertinent files into **c:/machines/ql/.**

## Updating the firmware

In ZX Spectrum Next terminology, *firmware* is the file called **TBBLUE.FW** that's located in the root folder of the SD card that holds your **System/Next™** distribution. It is impossible to start the machine without it, as it's a special program that configures all aspects of the machine regardless of personality and core. To update it, you only have to copy the new version over the previous TBBLUE.FW version. The current FW version is reported on the boot screen. See your *Quick Start Guide* to see how the core gets reported while booting.

## Updating the System/Next™ distribution

Every time a new version of *NextZXOS* with additional features gets released, it gets pushed to the System/Next git repository. Same thing happens with every software tool, firmware version and core that adds some feature or fixes a bug. A new System/Next™ will get released in a complete image form only when enough components have been updated as the process is very time consuming and only a large enough update on many components warrants this. So your system updates may be complete (ie. replacing all the components in the system in one go; firmware, core, operating system AND supporting tools) or just partial. You can update your System/Next™ distribution partially by going to the git repository at: gitlab.com/thesmog358/tbblue/ downloading the individual component and replacing it on your card. When updating *NextZXOS*, refer to *Chapter 19* to find out which files are absolutely required because they all need to be updated together.

That being said, NextZXOS attempts to make things easier for you by using an inbuilt "Updater" program. After downloading the appropriate / latest .DSU update file from www.specnext.com/latestdistro/ you need to place it on your SD card's root and then launch the *NextZXOS Startup Menu*, navigate to *More…* then go to *Tools* and select *Updater*. NextZXOS will locate the file and do everything for you!

Alternatively you can choose to download the entire distribution from git in one go by selecting the download button on the right top part of the distribution page.

If you do not feel adventurous however, the official home for the System/Next™ distribution is: www.specnext.com/latestdistro/ which also contains links to other forms of the distribution such as complete SD card images in various sizes for direct burning onto SD cards. Alternatively you have the option of purchasing a new SD card with the latest distribution on it from the SpecNext Ltd store.

## Selecting and configuring a personality

When powering up the system, you're presented with the boot screen, where, as we saw in the *Quick Start Guide* you're presented with the option of entering the Test Screen or to Press SPACEBAR for Menu.

Pressing SPACE (be quick or the option will disappear and booting will continue) will present you with the following screen:




*Fig. 52 – Personality Selection Screen*

By using the cursor keys and ENTER you can select a new personality which will then become your default one and all subsequent boots will get you into that. Selecting however

a personality and pressing **E** will allow you to configure the specific personality further. Doing so will present you with another screen:




*Fig. 53 – Configuration Options Screen*

Note that the screen is broken down into two parts, the top options are always used by NextZXOS while the options in the bottom part are ignored by NextZXOS that does its own enabling and disables/enables hardware configurations according to its needs.

There are a total of 15 personalities available and a few more may become available in a future update pending on core changes, two of which are Native Next modes; one with the standard 48K ROM and one with the Looking Glass 48K ROM which has the distinctive advantage of normal typing instead of tokenised entry. For Next Mode usage however both these are functionally equivalent and both provide access to dot commands in 48K mode.

The Pentagon 128K and TC2048 ones are the most idiosyncratic ones; the first operating only on 50Hz mode and was included to allow access to former Eastern-block countries' specially timed Spectrum Software and the TC2048 being the Timex Portugal partially Spectrum Compatible machine.

An *important thing* to remember is that for compatibility reasons the expansion bus is by default off; this doesn't mean you can plug in interfaces while the machine is working but that you will not have access to external peripherals unless you explicitly allow it via a series of **OUT** commands. This is to facilitate the usage of the onboard peripherals and the extra speed afforded by the Next's enhanced Z80n processor. All Next features are available in every mode unless you explicitly turn them off (so for example you need to turn off Timex modes via Configuration as above, if you don't want them) and you must install esxDOS yourselves (see relevant section in *Chapter 19* on how to do that) in order to access the onboard divMMC. Remember that the usage of external peripherals will slow down the machine personality to the 3.5MHz speed and only the onboard peripherals support the higher speeds. If you study *Chapter 23* and you know the specific ports your hardware uses, you can enable it yourself with a few easy command sequences.

Standard Sinclair BASIC lacks the **REG** command, so as seen in *Chapter 22* you will have to issue a series of **OUT** commands to enable external peripherals. For example to enable a ZX Printer (or Alphacom 32 or Timex Sinclair 2040) you will need to give:

```
OUT 9275, 136: OUT 9531, 219:
OUT 9275,128:OUT 9531,128
```

which disables the DACs on port FBh and immediately turns on the Expansion Bus. (You should however disable it afterwards so you can speed the machine up again).

A slightly different example is the following which enables the Interface 2. This time the relevant commands are:

```
OUT 9275, 128:OUT 9531,8:
OUT 9275,2 : OUT 9531,1
```

which does things a bit differently; first we select **NextREG 128** (**80h**) as before but this time we send it a value **8** which, as you can see from *Chapter 23*, is an instruction to enable the Expansion Bus after a soft reset and not immediately (setting bit 4). The last two **OUT**s are skipable because the soft reset they initiate can also be done by tapping on your **RESET** button for *less than 1 sec*.

## Troubleshooting

The Next team has taken every possible precaution and measure in order for your ZX Spectrum Next to live for a long time; inevitably however problems do arise. These are usually not related to the Next and the following paragraphs will hopefully assist you into figuring out quickly what potentially went wrong.

### If your screen is blank

- Check that your cables are connected and that your display is on and switched into that input and that your ZX Spectrum Next is powered.
- If the above are working check if you pressed **F3** by mistake or the program you're running has switched modes to a frequency your monitor doesn't support (eg. 60Hz). Press **NMI + 3** to switch frequencies.
- Verify you don't have a monitor that does 60Hz and you switched to Pentagon timings which only work at 50Hz. Reset the computer and press SPACE upon start to change personalities
- If you have a DVI monitor verify that your converter is working. Many HDMI to DVI converters do not work with the ZX Spectrum Next. Ask other users at the SpecNext forums for tested converters.
- If you connect your ZX Spectrum Next to a TV or an older CRT monitor via SCART, make sure that the line doubler feature is not turned on by mistake. Attempt to remedy by pressing **NMI + 2**.

### If you see a red screen

- Check the version of the core you're running if you see a message saying **Core 3.xx.yy required** and update your core.
- Check for a mismatched file versioning of *NextZXOS*. Prepare the SD card anew.
- If the above are okay, replace your SD card with a new card and repeat the process

### If your PS/2 keyboard is not working

- Check of in configuration mode, the PS/2 mode is set to **Keyboard**. **Core v.3.00** *and later* machines have this setting default to **Mouse**. If you want to use a keyboard, change this to **Keyboard** and if you want to use both, you will need to set this mode to Keyboard and purchase a Y-Splitter adapter, then plug the keyboard in its appropriate socket.

### Other things to look for

Other than the display not being able to support one of the display modes your machine may be in (which is approximately 90% of the cases), the other things to look for is connection/cable problems, SD card media failures or mis-configuration. As a general guideline, we suggest you first study the manual in the relevant sections, and if you still cannot figure out the problem, ask for help in SpecNext's forums, our Social Media accounts and the various groups online. If everything else fails, contact SpecNext Ltd and we'll try to find you a solution quickly!

# Appendix D

## The Calculator

The ZX Spectrum Next can be used as a full function calculator.

### Selecting the calculator

To use the calculator, call up the *Startup Menu* with **EDIT** and select the *Calculator* option. (If you don't know how to select a menu option, refer back to *Quick Start.)*

The calculator may be selected as soon as the ZX Spectrum Next is switched on.

Alternatively, if you are working on a *NextBASIC* program, you may select the calculator by choosing the *Exit* option from the *Edit/Options Menu* (which returns you to the *Startup Menu*), at which point you can select the *Calculator* option. Note that any *NextBASIC* program which was being worked on (when you selected the calculator) will be remembered and restored when you exit from the calculator and return to *NextBASIC*.

### Entering numbers

When you have selected the *Calculator* option, the screen will change to:

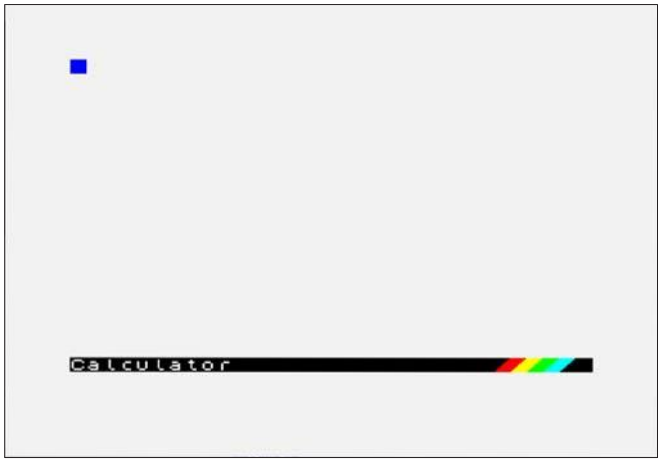



*Fig. 54 – Calculator Screen*

and the ZX Spectrum Next's calculator is ready to accept your first entry. Type in:

```
6+4
```

As soon as you press **ENTER**, the answer **10** will appear on the next line. (Note that you don't type = as you would on a conventional calculator.)

### Running total

You will see that the cursor is positioned to the right of the answer, which is a *running total* (like on a conventional calculator). This means that you can simply type in the next operation to be carried out on the running total (without having to type in a whole new calculation). So, with the cursor still positioned to the right of the **10** on the screen, type in:

```
/5
```

and the answer **2** appears.

## Using built-in mathematical functions

The ZX Spectrum Next's calculator leverages the power of *NextBASIC* to provide more advanced functions to the user. For example, with the result of the previous operation in place, type in:

```
*PI
```

This produces the result **6.2831853** on the screen. The ZX Spectrum Next has used its built-in π function – all you had to do was type in **PI**. This applies to all the ZX Spectrum Next's mathematical functions. To demonstrate, type in:

```
*ATN 60
```

which will give you the result **9.7648943**.

## Editing the screen

To further enhance the calculator's flexibility, you may also edit the contents of the screen. To demonstrate, move the cursor (using the cursor left key) to the beginning of the line and then type in **INT** so that the line reads

```
INT 9.7648943
```

and as soon as **ENTER** is pressed, the answer **9** will appear. This also demonstrates that the ZX Spectrum Next doesn't have to perform a calculation in order to print the value of an expression. As another example, press **ENTER** and type:

```
1E6
```

which will return the value of that expression. Notice that before you typed in **1E6**, you pressed **ENTER** on its own – this tells the ZX Spectrum Next that you are about to start a new calculation.

## Assigning variables

One extremely useful feature of the ZX Spectrum Next's calculator is that it allows you to assign values to variables and then use them in subsequent calculations. This is achieved by using the **LET** statement (unlike *NextBASIC* the Calculator doesn't yet allow assignments without **LET**). To demonstrate, press **ENTER** and type in the following:

```
LET x=10
```

You must then press **ENTER** twice for the ZX Spectrum Next to accept the variable assignment. Now verify that the variable **x** is being used, by typing:

```
x+90
```

then

```
+x*x
```

If you are using the calculator whilst working on a *NextBASIC* program, then any variables used by the calculator should be chosen so that they do not conflict with those used by the program itself. Note that *NextBASIC* keywords are not allowed to be used as variable names.

## User defined functions

Note that if you have set up any user defined functions (using the **DEF FN** statement) whilst working on a *NextBASIC* program, you will be able to invoke that function when using the calculator. To illustrate this point, return to *NextBASIC* and type in (for example):

```
9000 DEF FN c(n)=n*n*n
```

which sets up the user defined function **FN c(n)** which returns the *cube* of *n* (the number you type into the parentheses). Now exit from *NextBASIC* and return to the calculator – you can now use this user defined function as if it were one of the ZX Spectrum Next's own built-in functions. For example, enter:

```
FN c(3)
```

and the calculator will print the number **27** (i.e. the *cube of 3*).

## Exiting from the calculator

When you have finished using the calculator, press the **EDIT** key. The screen will change to:




*Fig. 55 – Calculator Options Menu*

Select the *Exit* option to return to the opening menu. If you were working on a *NextBASIC* program before you started using the calculator, then you may return to the program by selecting the *NextBASIC* option. (If you wish to continue using the calculator, then select the *Calculator* option).

# Acknowledgements


## Production - SpecNext Ltd

**Henrique Olifiers**
Creator, Director

**Mike Cadwallader**
Project Management

**Phoebus Dokos / Christina Stamatopoulou**
Distribution and Logistics

## Box

**Alfredo Tato**
Artwork

**Richard Hallas, Mike Cadwallader**
Copywriters

## Manual

**Phoebus Dokos**
Author

**Phoebus Dokos**
Artwork (content)

**Jonathan M Betts**
Artwork (cover)

**Dave Worton, Mike Cadwallader, Alvin Albrecht, Garry Lancaster, David Saphier, Matt Neale, John Kennedy, Will Stephenson, Rat Mal, FairFight14, Marc Kloosterman**
Additional & Crowdsourced Editing

## Coding

**Alvin Albrecht**
Spectrum Next Core
with core contributions by Mark Smith and Simon Brattel
Based on a design by Victor Trucco

**Garry Lancaster**
Firmware, NextZXOS, CP/M BIOS and Drivers, NextZXOS utilities and NextBASIC

**D Rimron-Soutter**
NextPi2

**Paul Farrow**
ZX80, ZX81 personalities

**Geoff Wearmouth**
Gosh Wonderful & Looking Glass ROMs

## Emulation / DevSystem

**#CSpect**
Mike Dailly

## Hardware

**Alvin Albrecht, Ignys**
Issue 4 Board
(Based on an original design by **Victor Trucco**)

## Software

**Kev Brady, Tim Gilberts, Tony Hoyle, David Saphier, Matt Davies, Robin Verhagen-Guest, D. Rimron, Garry Lancaster, Gari Biasillo, Simon N Goodwin, Simon Brattel, Neil Mottershead, Alvin Albrecht, César Hernández Bañó, Peter Helcmanovsky, Marco Varesio**

## KS 2 Games

**David Saphier**
Night Knight – based on the original by Juan J. Martínez

**Michael 'Flash' Ware, Lobo, Space Fractal**
Baggers in Space: The Detour

**Michael 'Flash' Ware, Simon Butler, Space Fractal**
Crowley: World Tour 2

**Michael 'Flash' Ware, Simon Butler, Paul Hesso**
Head Over Heels

## Hardware Support

**Robin Verhagen-Guest**
Wi-Fi, nxtp, NxTel

**Tim Gilberts**
UART, Mouse, RTC, I2C, Additional Wi-Fi support

**Mario Prato**
divMMC

## Industrial Design

**Phil Candy, Rick Dickinson**
Spectrum Next Case, Keyboard, Packaging

## Manufacturing

**Dave Worton**
Ever Sparkle Technologies

**Charlie Huxter, Angela Lennox**
Ceratech/Accuratus

## Testing

**Alvin Albrecht, Jim Bagley, Kev Brady, Mike Cadwallader, Phoebus Dokos, Tim Gilberts, Garry Lancaster, D Rimron-Soutter, David Saphier, Robin Verhagen-Guest, Simon N Goodwin**

## Special Thanks

**Alvin Albrecht, Gari Biasillo, Kev Brady, Mike Dailly, Matt Davies, Tim Gilberts, Garry Lancaster, Lampros Potamianos, D Rimron-Soutter, David Saphier, Lyndon J Sharp, Michael 'Flash' Ware**
For their unwavering drive and dedication in creating, enhancing, guiding and pushing the Spectrum Next forwards

**Our Kickstarter backers and shop purchasers**
For their belief and patience and for making the whole endeavour possible

## Additional Thanks

**Manuel Fernandez Higueras**
For creating the first fully compatible Spectrum Next board, the N-Go as well as testing and incorporating improvements to the platform

**Antonio Villena**
For creating the gomaDOS+ a ZX DOS board compatible with the Spectrum Next

**Don Superfo**
For creating the first ulta-mini compatible Next Board, the X-Berry π

**Evgeniy Barskiy, Dimitri Ponomarjov**
EnhancedULA ideas behind extra colour modes on layers 0 and 1

**David Banks**
BBC Core

# Table of Contents


**Chapter 1 – Basic Programming Concepts 5**
Introduction . . . 5
PRINT, LET, programs and line numbers . . . 5
Variables and Arrays . . . 6
Assignments . . . 6
Labels . . . 8
Using LIST, RUN and cursors to edit and run programs . . . 8
REM, NEW, INPUT and GO TO . . . 9
Using STOP, BREAK and CONTINUE . . . 10
Error trapping . . . 13
**ERROR** *[n]* . . . 14

**Chapter 2 – Decisions 16**
Making decisions . . . 16
Using IF to make decisions . . . 16
ELSE . . . 17
Case selection with ON . . . 19
Decisions with the ? operator . . . 19

**Chapter 3 – Looping 21**
Using FOR, TO and NEXT . . . 21
**STEP** . . . 22
EXIT . . . 23
REPEAT...REPEAT UNTIL loops . . . 24
WHILE . . . 25
Error trapping within REPEAT...REPEAT UNTIL loops . . . 26

**Chapter 4 – Procedures and Subroutines 27**
Branching . . . 27
GO SUB and RETURN . . . 27
LOCAL keyword . . . 28
PRIVATE and PRIVATE CLEAR . . . 29
Procedures (DEFPROC / ENDPROC / PROC) . . . 29
Passing parameters by reference with REF . . . 32
Reading parameters with DATA and READ . . . 33
Trapping errors locally . . . 33

**Chapter 5 – READ, DATA, RESTORE 35**
READ, DATA and RESTORE . . . 35
**DATA (function)** . . . 36

**Chapter 6 – Expressions 37**
Mathematical operations +, - , * , /, MOD . . . 37
Order of mathematical calculations . . . 37
Bitwise, relational and logical operators . . . 37
Unary/Bitwise NOT (!) . . . 38
Bitwise operators <<, >>, &, |, ↑, ↑| . . . 38
Expressions . . . 39
Variable names and limitations . . . 39
Scientific notation . . . 40
Decimal, Binary and Hexadecimal numbers . . . 41
More about Integer Expressions and Variables . . . 41
Signed vs Unsigned Integer Expressions . . . 45

**Chapter 7 – Strings 49**
Introduction . . . 49
String slicing, using TO . . . 49
String multiplication using the * operator . . . 50
Transforming a string with trailing modifers . . . 51
Tokenisation of strings . . . 51

**Chapter 8 – Functions 52**
Order of calculations using functions . . . 52
String functions . . . 53
LEN . . . 53
STR$ . . . 53
IN . . . 54
VAL and VAL$ . . . 55
Numeric functions . . . 55
SGN . . . 55
ABS . . . 56
**INT** . . . 56
SQR . . . 56
User defined functions using DEF and FN . . . 56
NextBASIC functions within integer expressions . . . 58

**Chapter 9 – Mathematical Functions 60**
↑ and EXP . . . 60
LN . . . 61
PI . . . 61
Trigonometry with SIN, COS, TAN, ASN, ACS and ATN . . . 62

**Chapter 10 – Random Numbers 64**
RANDOMIZE, RND and % RND . . . 64

**Chapter 11 – Arrays 67**
DIM . . . 67
DIM function . . . 70

**Chapter 12 – Conditions 71**
AND, OR and NOT . . . 71

**Chapter 13 – The Character Set 74**
CHR$ and CODE . . . 74
The graphics symbols . . . 74
Tokens . . . 75
BIN and USR . . . 75
POKE and PEEK . . . 76
Alternative Character Sets . . . 79
Character Graphics Mode . . . 80

**Chapter 14 – More about PRINT and INPUT 81**
Coordinate Systems . . . 81
Screen Modes and Pixel Coordinates . . . 81
Changing the size of characters . . . 82
Using AT to print to a certain location . . . 82
Using POINT to print to a certain location . . . 85
SCREEN$ . . . 88
TAB . . . 88
CLS . . . 89
Scrolling . . . 89
Expanding on INPUT . . . 90
LINE input . . . 91
Using Expressions for INPUT . . . 91
Using control codes with PRINT . . . 92
INKEY$ . . . 93
Using INPUT for game controllers . . . 93

**Chapter 15 – Colours 95**
An introduction to colour on the ZX Spectrum Next . . . 95
Basics of computer colour . . . 95
Colour organisation and representation . . . 95
Spatial vs Colour Resolution . . . 95
Colour attribute display . . . 98
Extended colour attribute display . . . 100
Palette-based hybrid linear bitmapped colour display . . . 101
Layer 3 colour storage . . . 102
Layer 2 priority colours . . . 102
More on the LAYER command . . . 103
BORDER, PAPER, INK, BRIGHT and FLASH . . . 105
BORDER . . . 107
INVERSE and OVER . . . 107
Using colour control codes . . . 107
ATTR . . . 108
PALETTE . . . 108

**Chapter 16 – Graphics 113**
PLOT . . . 113
DRAW and CIRCLE . . . 114
POINT, POINT TO . . . 116
Using OVER and INVERSE with graphics commands . . . 117
Using stippling patterns to generate additional colours . . . 118
Quick erase and fill using LAYER ERASE . . . 118
Clipping windows . . . 118
Tiling . . . 119
Accessing non-supported graphics modes . . . 120

**Chapter 17 – Time and Motion 121**
PAUSE . . . 121
Using POKE and PEEK at the System Variables . . . 122
TIME (command/function) . . . 122
Retrieving information from the RTC . . . 122
TIME$ . . . 123
INKEY$ . . . 125
Animation: a quick primer . . . 125
Mass Storage Frame Playback . . . 125
Memory Based Frame Playback . . . 127
Animation with the Sprite System . . . 128
Creating Sprites . . . 128
Putting Sprites on Screen . . . 130
Animating Sprites . . . 132
Moving Sprites on Screen . . . 134
Relative sprites . . . 136

Composite vs Unified sprites . . . 136
Batching . . . 137
Automatic sprite movement . . . 137
Sprite functions . . . 139
Scrolling . . . 140
The Copper . . . 140

**Chapter 18 – Sound and Music 144**
Basic sounds with the BEEP command . . . 144
Enhanced Sound and Music with PLAY . . . 147
Using the PLAY command . . . 147
Constructing strings . . . 148
PLAY command summary . . . 148
Setting the pitch . . . 148
Note duration . . . 149
The N Command . . . 151
Note volume . . . 151
Volume effects . . . 151
Tempo . . . 152
Repeated phrases . . . 152
The H command . . . 153
Comments . . . 153
Channel selection . . . 153
Stereo control . . . 153
Digital Audio . . . 153
WAV . . . 154
MOD . . . 154
PT3 . . . 154
Using the Pi accelerator for audio . . . 154
External Audio Output . . . 155

**Chapter 19 – NextZXOS and alternatives 157**
Guide to NextZXOS . . . 157
NextZXOS main features . . . 157
Files, Drives, Partitions and Disks . . . 158
Working with files . . . 158
Filenames . . . 159
LOAD . . . 160
SAVE . . . 164
VERIFY . . . 168
MERGE . . . 169
Using NextZXOS . . . 170
Wildcards . . . 170
Filesystems . . . 171
Partitions . . . 171
Storage devices and disks . . . 171
Mounting . . . 172
Drive cataloguing . . . 173
Drive, Folder and User Area navigation and management . . . 177
MKDIR . . . 178
RMDIR . . . 178
CD . . . 179
PWD . . . 180
Managing files and their attributes . . . 181
COPY . . . 181
ERASE . . . 182
MOVE . . . 183
File attributes . . . 184
The RAMdisk . . . 186
Drive and Partition Management . . . 186
CAT TAB and CAT ASN . . . 186
MOVE ... IN, MOVE ... OUT and REMOUNT . . . 187
Virtual filesystem management – .mkdata and .mkswap . . . 188
Printing . . . 189
The SPECTRUM command . . . 189
Speed Control . . . 191
NextBASIC Editor and Program support commands . . . 192
The Browser . . . 193
The Browser Window . . . 193
Using the Browser . . . 194
Configuring the Browser . . . 196
The Command Line . . . 196
ROM Cartridge Loaders . . . 197
48K BASIC . . . 197
128K BASIC . . . 197
ZX80 and ZX81 BASIC . . . 197
NMI Menu . . . 197
The NextZXOS folder structure . . . 199
NextZXOS dot commands . . . 199
Modifying the startup – Autoexec.bas . . . 200
CP/M . . . 201
Preparing your ZX Spectrum Next for esxDOS . . . 204

**Chapter 20 – Channels, Streams, Drivers and Windows 206**
Channels . . . 206
Streams . . . 208
Using Streams . . . 208
Stream control commands . . . 208
The Variable and Memory Channels . . . 211
Installable device drivers and Driver Channels . . . 213
Driver Channel support . . . 213
Windows . . . 214
System Windows vs User Windows . . . 214
User character sets . . . 216
Window input . . . 217
Window definitions . . . 217

**Chapter 21 – Optional Features 220**
Overview . . . 220
Installation (for Issue 2 mainboards) . . . 220
Raspberry Pi Zero installation on the Issue 4 mainboard . . . 222
Testing the add-ons' installation . . . 222
A. Testing the memory . . . 223
Using the Real Time Clock hardware . . . 227
Using the RTC together with the WiFi module . . . 228
Using the rest of the add-ons . . . 228

**Chapter 22 – IN, OUT and the Next Registers 229**
IN and OUT . . . 229
Hardware address decoding . . . 229
Accessing the ZX Spectrum Next features with NextREG . . . 232
The Next Registers . . . 234
The Expansion Bus . . . 246

**Chapter 23 – The Memory 247**
Overview . . . 247
ROM and RAM . . . 247
The Memory Map . . . 247
Memory Management . . . 248
Reading and Writing to Memory . . . 249
NextZXOS and NextBASIC memory allocation . . . 251
Memory Areas and their use . . . 252
NextBASIC Data Structures . . . 253
PEEK, POKE and their variants . . . 256
CLEAR . . . 259
Memory Bank management with BANK . . . 259
Using BANK with graphics . . . 262
Using BANK with files . . . 264
Extending NextBASIC Programs with BANK . . . 265
NextZXOS Paging Mechanism Overview . . . 265
MMU-Based Memory Management . . . 269
Layer 2 Bank Switching . . . 269
Paging method interactions . . . 269
Paging out the ROM . . . 269

**Chapter 24 – The System Variables 271**
Overview . . . 271
System Variables . . . 271

**Chapter 25 – Using Machine Code 275**
Using Machine Code . . . 275
Using CLEAR to Make Space . . . 275
Using USR to run machine code . . . 276
Calling NextZXOS from NextBASIC . . . 278
Opcodes Prefixes . . . 281

**Appendix A – Character Set, Z80N Mnemonics and Control Codes 282**

**Appendix B – Reference 288**
Reports and Error Codes . . . 288
General Errors . . . 288
Storage Device Related Errors . . . 290
NextBASIC Keywords and Functions . . . 291
The Decimal System . . . 295
The Binary System . . . 295
The Hexadecimal System . . . 295
Bits, Bytes and Words . . . 296
Using Binary and Hex in NextBASIC . . . 296

**Appendix C – Machine Personalities 297**
Overview . . . 297
The Cores and their update procedures . . . 297
Regular Core update . . . 298
AB Core update . . . 298
Multicore (Extra Cores) update . . . 299

Updating the firmware . . . . . 299
Updating the System/Next™ distribution . . . . . 300
Selecting and configuring a personality . . . . . 300
Troubleshooting . . . . . 302
Other things to look for . . . . . 302

Appendix D– The Calculator 303
Selecting the calculator . . . . . 303
Entering numbers . . . . . 303
Running total . . . . . 303
Using built-in mathematical functions . . . . . 304
Editing the screen . . . . . 304
Assigning variables . . . . . 304
User defined functions . . . . . 304
Exiting from the calculator . . . . . 305

Acknowledgements 307

Table Of Contents 309

Index 312

Technical Specifications 317
Issue 2 . . . . . 317
Issue 4 . . . . . 318

## !-0

.nxmod . . . . . 154
.playpt3 . . . . . 154
PLOT . . . . . 117
%RND . . . . . 64,65,66
%symbol . . . . . 42
%@ . . . . . 110
( TO ) . . . . . 49
.nexload . . . . . 154
> . . . . . 71
4-bit . . . . . 102
5 Out of screen, 0:1 . . . . . 83,85
512 colours . . . . . 95
8-bit . . . . . 95,101
9-bit . . . . . 95,101

## A

ABS. . . . . 56,58,59
ACS . . . . . 60,62,63
Alternative Character Sets . . . . . 79
anchor sprite . . . . . 136
AND . . . . . 38,44,71,72,73
Animating Sprites. . . . . 132,133
arccosine . . . . . 62
arcsine . . . . . 62
arctangent . . . . . 62
arguments. . . . . 52,58
array of pointers . . . . . 97
array variable . . . . . 67
Arrays. . . . . 67,68,69,70
ASCII. . . . . 49,74
ASN . . . . . 60,62,63
assignment. . . . . 5
AT . . . . . 82,83,84,89,90,92,113,120
ATN. . . . . 60,62,63
ATTR . . . . . 108,112
attribute cell . . . . . 96,100
Automatic sprite movement . . . . . 137,138

## B

B Integer out of range . . . . . 146
BANK . . . . . 30,43,80,97,109,110,126
BANK ERASE . . . . . 98
BANK DPEEK. . . . . 59
BANK PEEK. . . . . 59
BANK USR . . . . . 59
BANK USR$ . . . . . 59
BANK LAYER . . . . . 128
BANK POKE. . . . . 97,120,130
BANK PROC . . . . . 30
BANK RESTORE . . . . . 36
base10 logarithms . . . . . 61
Batching . . . . . 137
BEEP . . . . . 19,125,144,145,146,147,155
BIN . . . . . 59,75,76,110,141
Binary . . . . . 41
bitmask . . . . . 93
Bitwise AND. . . . . 44
Bitwise operators , >>, &, |, -, -| . . . . . 37, 38
Bitwise OR . . . . . 45
bitwise XOR. . . . . 45
Bitwise, relational and logical operators . . . . . 37
BORDER . . . . . 105,106,107
bounding box . . . . . 139
Branching. . . . . 27
BREAK. . . . . 25,90
BRIGHT . . . . . 100,102,105,106,109,115,120

## C

CAPS SHIFT . . . . . 74,90
Channel selection . . . . . 153
Channels . . . . . 123
Character Graphics mode . . . . . 97
character matrix . . . . . 82
CHARS system variable . . . . . 80
CHR$. . . . . 74,76,92
CIRCLE . . . . . 88,113,114,115,116,120
circumference . . . . . 62
CLEAR . . . . . 89,114
Clipping windows. . . . . 118,140
CLS . . . . . 89,114,141
CODE. . . . . 74
COL_FILE2 . . . . . 99
collision detection . . . . . 139
Colour attribute display. . . . . 98
Colour clash . . . . . 99,100
colour resolution . . . . . 96
COLOUR_FILE . . . . . 99
Colours . . . . . 95,96,97,98,99,100,101,102,103
. . . . . 104,105,106,107,108,109,110,111,112
Comments. . . . . 153
composite sprite . . . . . 136
conditional expression. . . . . 73
conditional selection . . . . . 36
Conditions . . . . . 71,72,73
CONTINUE . . . . . 23
control characters . . . . . 76
control codes. . . . . 76, 82,83,92
control variable . . . . . 22
coordinate system . . . . . 84
Coordinate Systems . . . . . 81
Copper . . . . . 121,142
COS . . . . . 60,62,63
COT. . . . . 63

## D

DACs . . . . . 155
DATA . . . . . 33,35,36,67,70,128
DATA (function) . . . . . 36
Decimal . . . . . 41
DEF FN . . . . . 56,57,61
DEFPROC . . . . . 29,30,31,33
degrees . . . . . 63
DELETE . . . . . 74
DIM . . . . . 67,68,69
DIM function . . . . . 70
dimension . . . . . 68
DISP_FILE . . . . . 97,98,99,100
DISP_FILE1 . . . . . 97,98,100
DISP_FILE2 . . . . . 97,98,99,100
dot command . . . . . 122
DPEEK . . . . . 59
DRAW . . . . . 88,113,114,115,116

## E

EDIT . . . . . 117
Edit menu . . . . . 89
ELSE . . . . . 16,17,18,19,20,36
ELSE IF . . . . . 18
END IF . . . . . 18,20
ENDPROC . . . . . 19,24,29,30,31,32,42
EnhancedULA . . . . . 81,82,100,101,106,109,115
ENTER . . . . . 65,91,125
Esc . . . . . 90
even-tempered tuning . . . . . 146
EXIT . . . . . 19,23,24
EXP . . . . . 60,61
Expansion . . . . . 142
Expressions . . . . . 39
Extended colour attribute display . . . . . 98

## F

false . . . . . 71,72
FLASH . . . . . 100,102,105,106,109,115,120
FN . . . . . 56,57
FOR . . . . . 21,22,23,24,42,114
FPGA . . . . . 128
FRAMES . . . . . 122
Full Ink Mode . . . . . 101
function . . . . . 44, 52

## G

G1R1B1 . . . . . 95
G3R3B2 . . . . . 95
GO SUB . . . . . 27
GO TO . . . . . 17,18,19,20,23,27,65
GPIO . . . . . 155
GRAPHICS . . . . . 74
graphics modes . . . . . 81
graphics symbols . . . . . 74
GRB . . . . . 95

## H

hardware scrolling . . . . . 140
Hexadecimal . . . . . 41
HiColour . . . . . 81,99,100
HiRes . . . . . 81,84,89,98,99,113
horizontal coordinates . . . . . 98
horizontal resolution . . . . . 96
horizontal size . . . . . 96
Hz . . . . . 121

## I

I2S . . . . . 155
IF . . . . . 16,17,20,71
IN . . . . . 54,58,80,101
INK . . . . . 75,99,100,105,106,109,115,120
INKEY$ . . . . . 93,125
INPUT . . . . . 18,21,23,42,58,67,81,84,90,91 . . . . . 93,106,115,125
INPUT function . . . . . 94
INPUT item . . . . . 90,91
INPUT LINE . . . . . 91
INPUT n . . . . . 93
INT . . . . . 56,57,64
INT {...} . . . . . 42,58
Integer arithmetic . . . . . 47
integer array . . . . . 67, 68
Integer Expressions . . . . . 37
interleaved storage . . . . . 96
INVERSE . . . . . 107,115,117,118,120

## K

K Invalid Colour . . . . . 109
keyboard joystick . . . . . 94

## L

Labels . . . . . 8
LAYER . . . . . 81,89,98,100,103,104,108,113,135
Layer 0 . . . . . 81,82,84,85,88,89,90,91,126
Layer 1 . . . . . 81,84,85
LAYER 1,0 . . . . . 88,92
LAYER 1,1 . . . . . 91,98
LAYER 1,2 . . . . . 89,98
LAYER 1,3 . . . . . 100
Layer 2 . . . . . 81,83,96,97,102,115
Layer 3 . . . . . 80,81,83,97
LAYER AT . . . . . 140
LAYER CLEAR . . . . . 105,119
LAYER DIM . . . . . 119
LAYER ERASE . . . . . 118,119
LAYER OVER . . . . . 103,104,118
LAYER PALETTE . . . . . 103,109,110,135
LAYER PALETTE BANK . . . . . 109

LEN . . . . . 53
LET . . . . . 5,16,42,61,117
LINE input . . . . . 91
literals . . . . . 41
LN . . . . . 60,61
LOAD ... LAYER . . . . . 127
LOCAL . . . . . 28,29,34,57,58
Localised error-trapping . . . . . 33,34
Logical expressions . . . . . 71
Logical operators . . . . . 38
LoRes . . . . . 81,84,102,103,104,106,114,115

## M

Mathematical Functions . . . . . 60,61,62,63
Mathematical operations . . . . . 37
MID$ . . . . . 58
MOD . . . . . 37,47,154
MOD files . . . . . 154
MOVE . . . . . 142
Moving Sprites . . . . . 134,135
MP3 files . . . . . 155
MSB . . . . . 102

## N

natural logarithm . . . . . 61
natural scale . . . . . 146
NEW . . . . . 50,114,119
NextREG . . . . . 140
NEXT . . . . . 21,22,23,24
NEXT # . . . . . 125
Next Register . . . . . 124
NextBASIC Menu . . . . . 117
NextZXOS . . . . . 88
NMI Menu . . . . . 134
NOT . . . . . 38,71,72,73
Note duration . . . . . 149,150
Note volume . . . . . 151
null string . . . . . 41
numeric arrays . . . . . 68

## O

octave command . . . . . 149
ON . . . . . 16,19,20,36
ON ERROR . . . . . 33
OR . . . . . 38,45,71,72,73
Order of calculations . . . . . 52
OUT . . . . . 80,101,141
OVER . . . . . 88,107,115,117,118

## P

PALETTE . . . . . 103,107,108,109,110,111,112,143
PALETTE CLEAR . . . . . 109
PALETTE DIM . . . . . 108,135
PALETTE FORMAT . . . . . 109
palettes . . . . . 100,101
PAPER . . . . . 75,99,100,105,106,109,115,120
parameters by reference . . . . . 32
PAUSE . . . . . 121,123,124
PAUSE/STOP . . . . . 142
PEEK . . . . . 58,76,77,78,80,122,125,128
PI . . . . . 61
pixels . . . . . 96,98
PI . . . . . 116
PLAY . . . . . 145,147,148,153,155
PLOT . . . . . 88,113,114,115,116,120
PLOT INVERSE . . . . . 117
PLOT OVER . . . . . 117
POINT . . . . . 84,85,86,87,91,113,116,117
POINT TO . . . . . 116,117
POKE . . . . . 76,77,78,79,80,92,93,97,122,128
predimensioned . . . . . 67
PRINT . . . . . 5,20,89,21,23,40,43,76,81
. . . . . 82,84,90,94,97,106,115
PRINT . . . . . 20,89
PRINT AT . . . . . 83,85,88
PRINT item . . . . . 82,83,85,90
PRINT POINT . . . . . 85
PRINT separator . . . . . 82
priority colours . . . . . 102
PRIVATE . . . . . 29
PRIVATE CLEAR . . . . . 29
PROC . . . . . 29,30,31,32,33,42
PROCedure . . . . . 112
Procedures . . . . . 29,30,31
Procrustean assignment . . . . . 50,69
PSGs . . . . . 144,149

## R

R3G3B2 . . . . . 95,101
R3G3B3 . . . . . 95
radians . . . . . 63
RAMdisk . . . . . 125,126
RAND . . . . . 65
random . . . . . 64
RANDOMIZE . . . . . 64,65,66
raster line . . . . . 141
READ . . . . . 33,35,36,42,70,130
recursion . . . . . 32
REF . . . . . 32,58
REG . . . . . 58,80,123,128,141
Relational operators , >, = ,=, >=, > . . . . . 39
relative sprite . . . . . 136,138
REPEAT . . . . . 23,24,25,26
REPEAT UNTIL . . . . . 23,24,25,26
resolution . . . . . 95
RESTORE . . . . . 35,36

RETURN . . . . . 19,24,27,29
RGB . . . . . 95
RIGHT$ . . . . . 58
RND . . . . . 59,64,65,66
RND () . . . . . 64,65
ROM . . . . . 88
row by column coordinate . . . . . 97
RTC . . . . . 122,125
RUN . . . . . 19,35,36,47,89,114,132,134

## S

SAVE . . . . . 130
SAVE ... BANK . . . . . 130
Scientific notation . . . . . 40
screen memory . . . . . 97
SCREEN$ . . . . . 88
Scrolling . . . . . 89,140
Scroll-prompt inhibitor . . . . . 92
SD . . . . . 126
SDH files . . . . . 155
SELECT CASE . . . . . 19
select operator ? . . . . . 16,19
semitone . . . . . 146
SGN . . . . . 55
SGN {...} . . . . . 39,47,48
SHIFT . . . . . 93
SID files . . . . . 154
Signed . . . . . 45,46,47,48
signed expression . . . . . 48
signed integer expressions . . . . . 47
SIN . . . . . 60,62,63
sine . . . . . 62
single-dimension character array . . . . . 70
SPACE . . . . . 90
spatial resolution . . . . . 96
SPRITE . . . . . 108,113,130,131,136,137
SPRITE AT . . . . . 139
SPRITE BANK . . . . . 130,131
SPRITE BORDER . . . . . 130,132
SPRITE CLEAR . . . . . 130
SPRITE CONTINUE . . . . . 137,139
SPRITE DIM . . . . . 132
Sprite functions . . . . . 139
Sprite Layer . . . . . 81
SPRITE MOVE . . . . . 137
SPRITE MOVE INT . . . . . 137
SPRITE OVER . . . . . 139
SPRITE PALETTE . . . . . 135
SPRITE PALETTE BANK . . . . . 135
SPRITE PAUSE . . . . . 139
sprite ports . . . . . 128
SPRITE PRINT . . . . . 130,131
SPRITE (Function) . . . . . 139
SPRITE STOP . . . . . 137
Sprite System . . . . . 101,128
SQR . . . . . 56,59
STEP . . . . . 22,23,26
Stereo control . . . . . 153
stippling . . . . . 118
STOP . . . . . 90
STR . . . . . 53
Streams . . . . . 123
String . . . . . 53
String functions . . . . . 53
String multiplication . . . . . 50
String slicing . . . . . 49
subexpression . . . . . 48
subscript . . . . . 67
subscripted variables . . . . . 67,68
subscripts . . . . . 69
substring . . . . . 58
SYMBOL SHIFT . . . . . 90
System Variables . . . . . 93,122

## T

TAB . . . . . 82,88,89,92
tabulating character . . . . . 88
TAN . . . . . 60,62,63
tangent . . . . . 62
Tempo . . . . . 152
Text windows . . . . . 82
The Copper . . . . . 140,141,142,143
THEN . . . . . 16,17
tile . . . . . 80
TILE . . . . . 113,119
TILE AT . . . . . 119
TILE BANK . . . . . 119
TILE DIM . . . . . 119
tile offset . . . . . 120
TILE w,h AT x,y . . . . . 119
TILE w,h AT x,y TO x2,y2 . . . . . 120
TILE w,h TO x2,y2 . . . . . 119,120
tilemap . . . . . 119
tiles . . . . . 119,120
TIME . . . . . 122,124
TIME$ . . . . . 123,124
Timex . . . . . 81
TL$ . . . . . 58
TO . . . . . 21,22,32,49
Tokenisation . . . . . 51
trailing modifers . . . . . 51
transparency . . . . . 130
transparency colour index . . . . . 109
transparency colour mask . . . . . 109

true . . . . . 71,72
two-dimensional array . . . . . 68
TZX files . . . . . 156

## U

UART . . . . . 155
UDG . . . . . 49,75,128
ULA . . . . . 103
unary ! . . . . . 48
unary not . . . . . 46
Unary/Bitwise NOT (!) . . . . . 38
unified sprite . . . . . 137,138
Unsigned . . . . . 45,46,47,48
unsigned expression . . . . . 48
user-defined graphic . . . . . 76
user-defined graphics . . . . . 75
Using Expressions for INPUT. . . . . 91
USR . . . . . 59,75,76
VAL . . . . . 51,52,55,124
VAL$ . . . . . 51,55
Variable names . . . . . 39
Variable not found . . . . . 92
vertical size . . . . . 96
visibility . . . . . 136
Visibility flag . . . . . 136
Volume effects. . . . . 151

## W

WAIT . . . . . 140,141
WAV . . . . . 154
WHILE. . . . . 25
WRITE . . . . . 142

## X

XOR . . . . . 45

# ZX Spectrum Next Home Computer

## Technical Specifications

### Issue 2 Model

- Xilinx Spartan-6™ SLX16 FPGA (XC6SLX16) implementing:
  - CPU:Z80N CPU with extended instruction set @ 3.5/7/14/28 MHz
  - All standard ZX Spectrum and Timex video-modes with the addition of Layer 2, Layer 3 and LoRes video with 9 bit colour and hardware scrolling
  - TurboSound compatibility (3 PSGs) - 3 x AY-3-8912 or YM-2149 compatible PSG audio chips with stereo output
  - Covox™/Soundrive™/SpecDrum™ compatible digital audio
  - Selectable DMA controller (Z80DMA and zxnDMA)
  - Amiga™ like Copper hardware
  - Hardware Sprite Engine
  - EnhancedULA extending legacy Spectrum and Timex modes to 256 colours out of a 512 colour palette
  - Multiface compatible NMI handling hardware
  - Two programmable UARTs
  - Programmable CTC chip (8 channels)
  - $I_2C$ bus
  - SPI bus
  - PS/2 keyboard and mouse controller
  - Two joystick controllers compatible with Kempston, Sinclair and Cursor standards
  - divMMC interface with an external SD card slot (as well as additional possibility for a secondary internal micro SD card slot, only via expert soldering at user's own risk)
- Memory:1MB SRAM (expandable to 2MB SRAM)
- 58 key laptop-style low profile tactile matrix keyboard
- Video out ports: RGB / VGA Analog and HDMI-compatible Digital Video Port
- Stereo Audio Out port
- Two multipurpose controller and I/O ports for joystick and serial communications
- Tape support, with joint Mic and Ear ports
- Original external bus expansion port
- Internal accelerator expansion port (for optional Raspberry Pi Zero[1] accelerator)
- Optional $I_2C$ RTC (Real Time Clock) device (DS-1307)
- Optional Wi-Fi module, with a full TCP/IP stack (ESP8266).
- On board GPIO for user expansion
- Multicore capability

1 *A Raspberry™ Pi Zero connects into the accelerator expansion port, which gives you one micro-USB port and an additional mini HDMI output. The Raspberry Pi Zero comes with a 1 GHz CPU, a GPU and 512 MB of RAM, and brings yet more possibilities to your ZX Spectrum Next, such as supporting a second display, additional sound playback and processing and even more advanced graphics processing power.*

## Issue 4 Model

- Xilinx Artix-7™ A15T FPGA (XC7A15T) implementing:
  - CPU:Z80N CPU with extended instruction set @ 3.5/7/14/28 MHz
  - All standard ZX Spectrum and Timex video-modes with the addition of Layer 2, Layer 3 and LoRes video with 9 bit colour and hardware scrolling
  - TurboSound compatibility (3 PSGs) - 3 x AY-3-8912 or YM-2149 compatible PSG audio chips with stereo output
  - Covox™/Soundrive™/SpecDrum™ compatible digital audio
  - Selectable DMA controller (Z80DMA and zxnDMA)
  - Amiga™ like Copper hardware
  - Hardware Sprite Engine
  - EnhancedULA extending legacy Spectrum and Timex modes to 256 colours out of a 512 colour palette
  - Multiface compatible NMI handling hardware
  - Two programmable UARTs
  - Programmable CTC chip (8 channels)
  - $I_2C$ bus
  - SPI bus
  - PS/2 keyboard and mouse controller
  - Two joystick controllers compatible with Kempston, Sinclair and Cursor standards
  - divMMC interface with an external SD card slot (as well as additional possibility for a secondary internal micro SD card slot, only via expert soldering at user's own risk)
- Memory:2MB SRAM standard
- 58 key laptop-style low profile tactile matrix keyboard
- Video out ports: RGB / VGA Analog and HDMI-compatible Digital Video Port
- Stereo Audio Out port
- Two multipurpose controller and I/O ports for joystick and serial communications
- Tape support, with joint Mic and Ear ports
- Original external bus expansion port
- $I_2C$ RTC (Real Time Clock) device (DS-1307)
- Wi-Fi module, with a full TCP/IP stack (ESP8266).
- Internal accelerator expansion port (for optional Raspberry Pi Zero accelerator)
- On board GPIO for user expansion
- Multicore capability